SQL Server Dapper

yuvalsol, 25 Jul 2016 CPOL

Rate:

★★★★★ 4.87 (22 votes)

Helper class to mitigate working with Dapper against SQL Server database

Download demo Dapper 1.42.0 - 54 KB

Download demo Dapper 1.50.2 - 545 KB

Download demo Dapper 1.50.2 .NET Core - 41 KB

Download SqlHelper (All versions) - 59 KB

Introduction

Dapper has emerged as one of the more powerful micro-ORMs out there. Its main focus is mapping database tables to POCOs (Plain Old CLR Objects) and it does that very quickly by all accounts. One of its strong points is its "genericness" to databases. Its inner implementation uses only DB interfaces such as `IDbConnection`, `IDbCommand`, `IDbTransaction`. There are no database-specific implementations (like `SqlConnection` which is SQL Server specific) so any data provider that implements these interfaces is good to go with Dapper. That strength can also be a bit of a weakness. Every database has its own little quirks and different ways of doing things and it's usually up to the user to write these things beforehand. A good example of a database-specific thing is SQL Server Table Valued Parameter or TVP for short. There are also some more common tasks, when working with data providers, that you want to encapsulate into a helper class. All these reasons and more were my motivation to write this helper class, the ubiquitous `SqlHelper`, which is designed specifically to mitigate working with Dapper against SQL Server database.

Connection String and Timeout

Before using this helper class in your project, you need to "massage" it a little bit. You need to edit the default implementation of `GetConnectionString` and `GetTimeout` methods to reflect the needs of **your** specific project, so you need to work them to your liking.

The ability to pass different connection string and connection timeout to a query method is still there, but as a rule of thumb you don't want to do that every time. You want to encapsulate this task for all your calls to the database and these methods are there for that. The default connection timeout that I use is 30 seconds. Feel free to change that too.

Hide Copy Code

```
public static class SqlHelper
{
    public static string GetConnectionString(string connectionString = null)
    {
        // this is where you write your implementation
        if (!string.IsNullOrEmpty(connectionString))
            return connectionString;

        return ConfigurationManager.ConnectionStrings["connection_string_name"].ConnectionString;
```

```
    }

    public static int ConnectionTimeout { get; set; }

    public static int GetTimeout(int? commandTimeout = null)
    {
        if (commandTimeout.HasValue)
            return commandTimeout.Value;

        return ConnectionTimeout;
    }
}
```

DynamicParameters - Using SQL Server Types

Dapper uses `DbType` enumeration to define the type of the parameter that you're passing to your SQL or stored procedures. Most of the time, you are not doing it explicitly. You just pass an anonymous object as parameter values and Dapper does the value-to-type mapping itself. However, there are times when you want to explicitly set the type of the parameter. This usually occurs when passing an output parameter to a stored procedure. Dapper uses `DynamicParameters` class to define a collection of parameters.

Hide Copy Code

```
// Dapper
partial class DynamicParameters
{
    public void Add(string name, object value = null, DbType? dbType = null,
    ParameterDirection? direction = null, int? size = null) { }
}

// Adding output parameters
DynamicParameters params = new DynamicParameters();
params.Add("ResultCode", dbType: DbType.Int32, direction: ParameterDirection.Output);
params.Add("ResultDesc", dbType: DbType.String, direction: ParameterDirection.Output, size: 4000);
```

The thing is, you don't want to figure out every time what is the mapping between your SQL Server type to `DbType` enumeration. You just want to use SQL Server types and that enumeration is `SqlDbType`. With the help of this MSDN page SQL Server Da... Type Mappings, I added internal mapping between the two enumerations and added a method overload to Dapper's `DynamicParameters` class. As a convenience, I also added non-default constructor to `DynamicParameters`, which I felt was missing.

Hide Shrink Copy Code

```
partial class DynamicParameters
{
    static readonly Dictionary<SqlDbType, DbType?> sqlDbTypeMap = new Dictionary<SqlDbType, DbType?>
    {
        {SqlDbType.BigInt, DbType.Int64},
        {SqlDbType.Binary, DbType.Binary},
        {SqlDbType.Bit, DbType.Boolean},
        {SqlDbType.Char, DbType.AnsiStringFixedLength},
        {SqlDbType.DateTime, DbType.DateTime},
        {SqlDbType.Decimal, DbType.Decimal},
        {SqlDbType.Float, DbType.Double},
        {SqlDbType.Image, DbType.Binary},
        {SqlDbType.Int, DbType.Int32},
        {SqlDbType.Money, DbType.Decimal},
        {SqlDbType.NChar, DbType.StringFixedLength},
        {SqlDbType.NText, DbType.String},
        {SqlDbType.NVarChar, DbType.String},
        {SqlDbType.Real, DbType.Single},
        {SqlDbType.UniqueIdentifier, DbType.Guid},
        {SqlDbType.SmallDateTime, DbType.DateTime},
        {SqlDbType.SmallInt, DbType.Int16},
        {SqlDbType.SmallMoney, DbType.Decimal},
        {SqlDbType.Text, DbType.String},
        {SqlDbType.Timestamp, DbType.Binary},
        {SqlDbType.TinyInt, DbType.Byte},
        {SqlDbType.VarBinary, DbType.Binary},
        {SqlDbType.VarChar, DbType.AnsiString},
```

```
        {SqlDbType.Variant, DbType.Object},
        {SqlDbType.Xml, DbType.Xml},
        {SqlDbType.Udt,(DbType?)null}, // Dapper will take care of it
        {SqlDbType.Structured,(DbType?)null}, // Dapper will take care of it
        {SqlDbType.Date, DbType.Date},
        {SqlDbType.Time, DbType.Time},
        {SqlDbType.DateTime2, DbType.DateTime2},
        {SqlDbType.DateTimeOffset, DbType.DateTimeOffset}
    };

    // non-default constructor with DbType
    public DynamicParameters(string name, object value = null, DbType? dbType = null,
    ParameterDirection? direction = null, int? size = null) : this()
    {
        Add(name, value, dbType, direction, size);
    }

    // overload constructor with SqlDbType
    public DynamicParameters(string name, object value = null, SqlDbType? sqlDbType = null,
    ParameterDirection? direction = null, int? size = null) : this()
    {
        Add(name, value, sqlDbType, direction, size);
    }

    // add parameter with SQL Server type
    public void Add(string name, object value = null, SqlDbType? sqlDbType = null,
    ParameterDirection? direction = null, int? size = null)
    {
        Add(name, value, (sqlDbType != null ?
        sqlDbTypeMap[sqlDbType.Value] : (DbType?)null), direction, size);
    }
}
```

You can see from the mapping that I didn't mapped two types, `SqlDbType.Udt` and `SqlDbType.Structured` to any specific `DbType`. Obviously, there is no straightforward mapping here.

`SqlDbType.Udt` is used for things like User-defined Types and more advanced types like `Geography`, `Geometry` and `HierarchyId`.

`SqlDbType.Structured` is used for Table-valued parameters when you pass a `DataTable` as value.
I simply mapped them to a `null DbType` and let Dapper sort it out.

DynamicParameters - Getting All Parameter Values

`DynamicParameters` has a `Get` method which simply returns the value of a parameter given its name. Its usage is usually when you retrieve the value from an output parameter. But just to be clear here, there is no limitation on the direction of the parameter.

You can get the value from an input parameter but obviously that won't make much sense.

Hide Copy Code

```
// Dapper
partial class DynamicParameters
{
    public T Get<T>(string name) { }
}

// Adding output parameters
DynamicParameters params = new DynamicParameters();
params.Add("ResultCode", dbType: DbType.Int32, direction: ParameterDirection.Output);
params.Add("ResultDesc", dbType: DbType.String, direction: ParameterDirection.Output, size: 4000);

....

int resultCode = params.Get<int>("ResultCode");
string resultDesc = params.Get<string>("ResultDesc");
```

When using multiple output parameters, usually you want to get all their values out and then work with all of them.

I added a couple of `Get` methods for that to pull all the values at once into a `Dictionary` instance. They use the underline `DynamicParameter`s' `Get` method for retrieving each parameter value. The first `Get` method returns a `Dictionary<string, object>` which maps the name of the parameter to its value. If you know that all the parameters are of the same type, you can use the generic version of `Get<T>` to retrieve a type-safe `Dictionary<string, T>`.

Hide Copy Code

```
partial class DynamicParameters
{
    public Dictionary<string, object> Get()
    {
        return Get<object>();
    }

    // all the parameters are of the same type T
    public Dictionary<string, T> Get<T>()
    {
        Dictionary<string, T> values = new Dictionary<string, T>();
        foreach (string parameterName in ParameterNames)
            values.Add(parameterName, Get<T>(parameterName));
        return values;
    }
}
```

The problem with this code is that it also gets the values of Input parameters which usually you don't want.
Later on, I will show how this problem is solved and how I differentiate between Input parameters and non-Input parameters.

Query

The Query methods wrap around Dapper's own query methods. Internally, they just make a call to a Dapper query method and return whatever it returns.

Conceptually, they mirror the `Dapper` method that they call, mostly by the type of the return value, however, they are structured a little bit different from `Dapper`'s methods.

Hide Copy Code

```
// stored procedure -  dynamic
IEnumerable<dynamic> QuerySP(string storedProcedure, dynamic param, dynamic outParam,
SqlTransaction transaction, bool buffered, int? commandTimeout, string connectionString) { }

// sql - dynamic
IEnumerable<dynamic> QuerySQL(string sql, dynamic param, dynamic outParam,
SqlTransaction transaction, bool buffered, int? commandTimeout, string connectionString) { }

// stored procedure - T
IEnumerable<T> QuerySP<T>(string storedProcedure, dynamic param, dynamic outParam,
SqlTransaction transaction, bool buffered, int? commandTimeout, string connectionString) { }

// sql - T
IEnumerable<T> QuerySQL<T>(string sql, dynamic param, dynamic outParam,
SqlTransaction transaction, bool buffered, int? commandTimeout, string connectionString) { }
```

First thing to notice is that the methods are separated to stored procedure methods and pure SQL methods. The stored procedure query methods end with "`SP`" and the pure SQL methods end with "`SQL`", so the query methods will be "`QuerySP`" and "`QuerySQL`".

This naming pattern will repeat itself throughout the rest of the helper class. The reasoning is two-fold. First, it separates the two distinct types of operation - stored procedure and pure SQL - into semantically two types of methods and help the user to differentiate between them. Although you can execute a stored procedure as pure SQL ("`exec sp_do_something`"), you really don't want to do that so as to prevent SQL Injection.

Second, the user doesn't have to pass the command type (`CommandType.Text`, `CommandType.StoredProcedure`) as a parameter since it is taken care of within these methods.

Another main difference is the fact that they take two sets of parameters, `param` and `outParam`. `param` takes a set of input parameters and `outParam` takes a set of non-input parameters (output, input-output, return value parameters).

This distinction is only semantical and up to the user to keep it. There's nothing to prevent you from passing one type of SQL

parameter through the other method argument. So why do that? If you execute a query directly through `Dapper`, you can see that the input parameters are almost always passed as an anonymous type, i.e., new { Name = value, ... } and non-input parameters are passed by declaring an instance of `DynamicParameters`. `DynamicParameters` is `Dapper` implementation of set of parameters.

Internally, `Dapper` takes the anonymous type that you pass as a set of input parameters and builds an instance of `DynamicParameters` from them. I'm not suggesting that you cannot define input parameters directly through `DynamicParameters`, however it is not the modus operandi for input parameters.

The reason that we take two seemingly separate ways of defining parameters is because we want to read the results from the non-input parameters but don't care about the input parameters. That is the reason why these methods have two sets of parameters, input and non-input. In cases when you want to pass a mixture of input and non-input parameters, you can still pass an anonymous set of parameters through `param` and an instance of `DynamicParameters` through `outParam`. After the query ends, you can get all the values, and only the values of the non-input parameters by calling `Get<T>()` as described previously.

The last difference is the type of transaction. The `Query` methods require that you pass a `SqlTransaction`, which is the SQL Server provider's implementation of `IDbTransaction`. `Dapper` only asks for `IDbTransaction`.

Query Examples

The examples query the `Production.Product` table from `AdventureWorks` database.

First we declare our POCO.

Hide Copy Code

```
// Product POCO
public class Product
{
    public int ProductID { get; set; }
    public string Name { get; set; }
}
```

Queries using pure SQL and stored procedure.

Hide Cop

```
// pure SQL
var products = SqlHelper.QuerySQL<Product>("select * from Production.Product");
var product = SqlHelper.QuerySQL<Product>("select * from Production.Product _
where ProductID = @ProductID", new { ProductID = 1 }).FirstOrDefault();

// stored procedure
var products = SqlHelper.QuerySP<Product>("sp_get_products");
var product = SqlHelper.QuerySP<Product>("sp_get_products", new { ProductID = 1 }).FirstOrDefault();
```

Queries with output parameters.

Hide Shrink Copy Code

```
// output parameters
// the parameter types are defines by SqlDbType enumeration, not by DbType enumeration
var outParam = new DynamicParameters("ProductsCount", sqlDbType: SqlDbType.Int,
    direction: ParameterDirection.Output);
outParam.Add("DummyInOutParam", 10, SqlDbType.Int, ParameterDirection.InputOutput);

// pure SQL and output parameters
var products = SqlHelper.QuerySQL<Product>(@"
    select @ProductsCount = count(*) from Production.Product;
    set @DummyInOutParam = @DummyInOutParam * 2;
    select * from Production.Product",
outParam: outParam);

var product = SqlHelper.QuerySQL<Product>(@"
    select @ProductsCount = count(*) from Production.Product;
    set @DummyInOutParam = @DummyInOutParam * 2;
    select * from Production.Product where ProductID = @ProductID",
new { ProductID = 1 }, outParam).FirstOrDefault();
```

```
// get specific output parameter value
int productsCount = outParam.Get<int>("ProductsCount");
int dummyInOutParam = outParam.Get<int>("DummyInOutParam");

// get all output parameter values
Dictionary<string, object> outValues = outParam.Get();
int productsCount = (int)outValues["ProductsCount"];
int dummyInOutParam = (int)outValues["DummyInOutParam"];

// both output parameters are of the same type int
Dictionary<string, int> outValues = outParam.Get<int>();
int productsCount = outValues["ProductsCount"];
int dummyInOutParam = outValues["DummyInOutParam"];
```

TVPs - Table Valued Parameters - A Quick Recap

TVPs were introduced with SQL Server 2008. They provide an easy way to pass multiple rows of data to SQL Server. TVP "acts" like (but is not) a table, meaning you can select from it just like you select from a table.
TVP must be declared in the stored procedure list of arguments as read-only so you cannot change its values, just read them. A TVP value is a `DataTable`. The `DataTable`'s columns have to match exactly that of the declaration of the TVP by name, type and order.

Before SQL Server 2008, you needed to resolve yourself to various hacks to do the same functionality that TVPs provide. A common problem is passing a list of numbers, such as indices, to SQL Server. In that case, a common hack would be to pass a list of values as a `string`, separated by a comma and then split the `string` on the SQL Server side (`dbo.fnSplit`) and convert the values into numbers.

Here's an example of how a `TVP` solves this problem. First, creating the `TVP`.

Hide Copy Code

```
CREATE TYPE IntArray AS TABLE
(
    IntValue int
)
```

Then declaring a `TVP` in a stored procedure:

Hide Copy Code

```
CREATE PROCEDURE [dbo].[sp_get_products]
    @ProductIDs IntArray readonly
AS
BEGIN
    SET NOCOUNT ON;

    select *
    from Production.Product
    where ProductID in (select IntValue from @ProductIDs)

END
```

Finally, creating a `DataTable` as the `TVP` value and creating a `TVP`.

Hide Copy Code

```
DataTable intArray = new DataTable();
intArray.Columns.Add("IntValue", typeof(int));
intArray.Rows.Add(1); // product 1
intArray.Rows.Add(2); // product 2
intArray.Rows.Add(3); // product 3

// SqlParameter with Structured sql type
SqlParameter sqlParameter = new SqlParameter();
sqlParameter.ParameterName = "ProductIDs";
sqlParameter.SqlDbType = SqlDbType.Structured;
sqlParameter.Value = intArray;
```

TVPs

Starting from version 1.26 Dapper supports TVPs. All you need to do is pass a `DataTable` as a value and `Dapper` will create the appropriate parameter.

Continuing the previous example...

Hide Copy Code

```
var products = SqlHelper.QuerySP("sp_get_products", new { ProductIDs = intArray });
```

You can also pass on the name of the TVP if you like. `AsTableValuedParameter` is an extension to `DataTable`, defined in `Dapper`, which uses `DataTable`'s `ExtendedProperties` to define the name of the TVP type. In this case, the name of the TVP type is "`IntArray`".

Hide Copy Code

```
var products = SqlHelper.QuerySP("sp_get_products",
new { ProductIDs = intArray.AsTableValuedParameter("IntArray") });
```

IEnumerable To DataTable

The problem with a TVP is that its value is a `DataTable`. You need to specifically construct a `DataTable` for that kind of parameter. That task gets tedious when you are working in a POCO-oriented environment. Such working environment also has the benefit of using LINQ to Objects. In an environment that holds data mainly with `IEnumerable` (like `Array`, `List<>`) and not with `DataTable`, building a `DataTable` each time you want to use a TVP becomes a monotonous task.

This is where the `SQLHelper` comes in play. `SQLHelper` has method extensions for `IEnumerable` and `Type` that return a `DataTable`. That gives you a seamless way to create TVP values very quickly and without the need to write that code yourself. The `IEnumerable` method extension is used when you want to create a TVP with some rows with it. The `Type` method extension is used when you want to create an empty TVP. The method extensions are declared as:

Hide Copy Code

```
DataTable ToDataTable<T>(this IEnumerable<T> instances, string typeName,
    MemberTypes memberTypes = MemberTypes.Field | MemberTypes.Property, DataTable table) { }

DataTable ToDataTable(this Type type, string typeName,
    MemberTypes memberTypes = MemberTypes.Field | MemberTypes.Property) { }
```

Before we delve into the implementation and the various options, let's look at a simple example. As before, we are working with `Product` POCO. Its TVP declaration is as follows:

Hide Copy Code

```
CREATE TYPE ProductTVP AS TABLE
(
    ProductID int,
    Name nvarchar(50)
)
```

and its POCO class is:

Hide Copy Code

```
public class Product
{
    public int ProductID { get; set; }
    public string Name { get; set; }
}
```

Passing list of `products` as parameter to the stored procedure:

Hide Copy Code

```
var products = new List<Product>() {
    new Product() { ProductID = 1, Name = "Product 1" },
```

```
    new Product() { ProductID = 2, Name = "Product 2" },
    new Product() { ProductID = 3, Name = "Product 3" }
};

// convert a list of products to a DataTable of products
// and pass it as a TVP value
DataTable productsDT = products.ToDataTable<Product>();
var results = SqlHelper.QuerySP("sp_get_products", new { Products = productsDT });

// in one line
var results = SqlHelper.QuerySP("sp_get_products", new { Products = products.ToDataTable<Product>() });

// and pass the name of the TVP type
var results = SqlHelper.QuerySP("sp_get_products",
new { Products = products.ToDataTable<Product>("ProductTVP") });
```

The `IEnumerable` method extension is used when you want to create a `TVP` with some rows with it. The `Type` method extension is used when you want to create an empty `TVP`.

Hide Copy Code

```
DataTable ToDataTable<T>(this IEnumerable<t> instances, string typeName,
    MemberTypes memberTypes = MemberTypes.Field | MemberTypes.Property, DataTable table) { }

DataTable ToDataTable(this Type type, string typeName,
    MemberTypes memberTypes = MemberTypes.Field | MemberTypes.Property) { }    </t>
```

The `typeName` argument is the name of the `TVP` type in SQL Server.

The `table` argument in the first method extension is intended for appending rows to an existing `DataTable` that is passed to the method.

The `memberTypes` argument indicates what type of members in the POCO class we want to map to the `DataTable`. The default behavior is to map both the fields and the properties in the POCO. Normally, a POCO will be comprised of properties alone but the method gives extra wiggle room to work with fields too. The method extensions will take only a `public` field and a `public get` property (regardless of the `set` property exposure).

There is no guarantee what would be the order of fields and/or properties that are mapped to the `DataTable`. The intention is obviously the order of which they are declared in the code. Intuitively, we would want the order to be the sam order of which we wrote the POCO class. However this is not possible because there is no "declaration order" in `System.Reflection`.

That being said, and the way the extension methods are implemented, the order is not random as you might think. There is something in `System.Reflection` that can give out an impression of "declaration order" although it is not. The data members are ordered by MemberInfo.MetadataToken Property which uniquely identifies a metadata element. The `int` values that are assigned to `MemberInfo.MetadataToken` are usually (but definitely not guaranteed to be) ordered respectfully to the order of first, the declaration order of the fields and second, the declaration order of the properties.

The order of the following POCO's data members:

Hide Copy Code

```
public class Product
{
    public int Width;
    public int ProductID { get; set; }
    public int Height;
    public string Name { get; set; }
}
```

will be: `Width`, `Height`, `ProductID`, `Name`. Fields first, properties second. However, this example is contrived. Normally a POCO will be written solely with properties and since `MemberInfo.MetadataToken` is (usually) ordered by declaration order, the POCO data members will ordered correctly as columns in the `DataTable`.

This is the code snippet that implements all that was discussed here.

Hide Shrink Copy Code

```
private static DataTable ToDataTable (IEnumerable instances, Type type,
    string typeName, MemberTypes memberTypes, DataTable table)
{
```

```
    bool isField = ((memberTypes & MemberTypes.Field) == MemberTypes.Field);
    bool isProperty = ((memberTypes & MemberTypes.Property) == MemberTypes.Property);

    var columns =
        type.GetFields(BindingFlags.Public | BindingFlags.Instance)
            .Where(f => isField)
            .Select(f => new
            {
                ColumnName = f.Name,
                ColumnType = f.FieldType,
                IsField = true,
                MemberInfo = (MemberInfo)f
            })
            .Union(
                type.GetProperties(BindingFlags.Public | BindingFlags.Instance)
                    .Where(p => isProperty)
                    .Where(p => p.CanRead) // has get property
                    .Where(p => p.GetGetMethod(true).IsPublic) // is public property
                    .Where(p => p.GetIndexParameters().Length == 0) // not an indexer
                    .Select(p => new
                    {
                        ColumnName = p.Name,
                        ColumnType = p.PropertyType,
                        IsField = false,
                        MemberInfo = (MemberInfo)p
                    })
            )
            .OrderBy(c => c.MemberInfo.MetadataToken);
    ....
}
```

If you don't want to rely on `MemberInfo.MetadataToken` and want to make sure that the columns are in the right order, SetOrdinal is an extension method to `DataTable` that sets the ordinal number of the columns based on the order that you pass their names to the method.

Hide Copy Code

```
// set the columns ordinal numbers
public static void SetOrdinal(this DataTable table, params string[] columnNames) { }

// usage
table.SetOrdinal("Width", "ProductID", "Height", "Name");
```

DataTable To IEnumerable

Strictly speaking, there is no need here for conversion from `DataTable` to any kind of `IEnumerable`. Dapper return values are `IEnumerable`. However, this is a helper class, and since I already implemented conversion of `IEnumerable` to `DataTable`, I also added the other way around for the sake of completeness. I might also say, out of my own experience, that in a development environment which uses **both** POCOs and `DataTable`s as business objects to carry data around, the ability to convert at ease a POCO from **and** to a `DataTable` is very helpful.

The first 3 methods below - `Cast<T>()`, `ToArray<T>()`, `ToList<T>()` - are generic methods with type `T` which has a default constructor. In cases when `T` doesn't have a default constructor, the next 3 methods will come in handy. You can pass as argument a delegate that returns an instance of type `T`.

There are also 6 more methods with the same templates except for a `DataView` instead of a `DataTable`.

Hide Copy Code

```
// T has a default constructor
IEnumerable<T> Cast<T>(this DataTable table) where T : new() { }
T[] ToArray<T>(this DataTable table) where T : new() { }
List<T> ToList<T>(this DataTable table) where T : new() { }

// T is instantiated with a delegate
IEnumerable<T> Cast<T>(this DataTable table, Func<T> instanceHandler) { }
T[] ToArray<T>(this DataTable table, Func<T> instanceHandler) { }
List<T> ToList<T>(this DataTable table, Func<T> instanceHandler) { }
```

What happens when the types are not the same? The table column could be one type and the POCO property can be another type. By default, the methods use `Convert.ChangeType()` to enforce the POCO property type. For example, the table column is `decimal` and the POCO property is `int`. `Convert.ChangeType()` will convert the table value from decimal to int. This is a reasonable solution when the types can be converted from one to the other. If the types can't be converted (`Convert.ChangeType()` throws an exception) or in cases when you would like to do your own manipulation on the value before it is assigned to the POCO property, you can pass a delegate, as argument to any of the methods above, which takes the original value and returns another value.

Hide Copy Code

```
IEnumerable<T> Cast<T>(this DataTable table, ValueHandler getValue) { }
T[] ToArray<T>(this DataTable table, ValueHandler getValue) { }
List<T> ToList<T>(this DataTable table, ValueHandler getValue) { }
```

This example will convert the `decimal` values to string but their `string` representation will have 4 digits after the decimal symbol.

Hide Copy Code

```
public class POCO
{
    public string Number { get; set; } // string type
}

DataTable table = new DataTable();
table.Columns.Add("Number", typeof(decimal)); // decimal type

table.Rows.Add(1);
table.Rows.Add(2.3);
table.Rows.Add(4.56);
table.Rows.Add(7.891);

IEnumerable<POCO> list = table.Cast<POCO>(delegate(string name, Type type, object value)
{
    if (name == "Number")
        return ((decimal)value).ToString("N4");
    return Convert.ChangeType(value, type);
});
```

QueryMultiple

So how does `Dapper` handle returning multiple data sets? It uses an internal class called `GridReader` which you can read from it the next available data set until the `GridReader` is consumed. The `QueryMultiple` methods simply wrap around this `GridReader` and return a list of lists or a Tuple of lists. That's all.

If the multiple datasets, that return from SQL Server, have the same structure, meaning they map to the same POCO, the `QueryMultiple` method will return `IEnumerable<IEnumerable<T>>`. If the multiple datasets have different structure from each other, the `QueryMultiple` method will return a Tuple or lists. For 2 datasets, of type `T1` and `T2`, it'll return `Tuple<IEnumerable<T1>, IEnumerable<T2>>`.
There are `QueryMultiple` methods from 2-Tuple up to 14-Tuple (the 8th item in a 8-Tuple is a Tuple itself, thus extending the Tuple the 8 items and more).

Hide Copy Code

```
// all the enumerables are of the same type T
IEnumerable<IEnumerable<T>> QueryMultipleSQL<T>(string sql, dynamic param, dynamic outParam,
    SqlTransaction transaction, int? commandTimeout, string connectionString) { }

// 2 enumerables, T1 and T2. can be different types
// there are method overloads from 2-Tuple up to 14-Tuple
Tuple<IEnumerable<T1>, IEnumerable<T2>> QueryMultipleSQL<T1, T2>(
    string sql, dynamic param,
    dynamic outParam, SqlTransaction transaction,
    int? commandTimeout, string connectionString
) { }
```

QueryMultiple To DataSet

We already saw that we can turn an `IEnumerable` to a `DataTable`. So if we have multiple `IEnumerable` instances, we can turn each to a `DataTable` and bundle them all together in a `Dataset`.

`ToDataSet` extension methods do just that.

Hide Copy Code

```
// all the enumerables are of the same type T
DataSet ToDataSet<T>(this IEnumerable<IEnumerable<T>> instances,
    string[] typeNames, MemberTypes memberTypes = MemberTypes.Field | MemberTypes.Property,
    DataSet dataSet) { }

// 2 enumerables T1 and T2. can be different types
// there are method overloads from 2-Tuple up to 14-Tuple
DataSet ToDataSet<T1, T2>(this Tuple<IEnumerable<T1>, IEnumerable<T2>> instances,
    string[] typeNames, MemberTypes memberTypes = MemberTypes.Field | MemberTypes.Property,
    DataSet dataSet) { }
```

QueryMultiple Examples

The first example returns 2 results, both are `Production.Product`.

Hide Copy Code

```
// Product POCO
public class Product
{
    public int ProductID { get; set; }
    public string Name { get; set; }
}
```

Return multiple results with `QueryMultipleSQL`.

Hide Copy Code

```
var products = SqlHelper.QueryMultipleSQL<Product>(@"
    select * from Production.Product where Product_Code <= 10;
    select * from Production.Product where Product_Code > 10;
");
```

And convert the results to a `DataSet`.

Hide Copy Code

```
DataSet productsDS = products.ToDataSet<Product>();
```

The second example returns 2 results, one result is `Production.Product` and the other one is `Person.Person`, and convert the results into a `DataSet`.

Hide Copy Code

```
// Person POCO
public class Person
{
    public int BusinessEntityID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

DataSet resultsDS = SqlHelper.QueryMultipleSQL<Product, Person>(@"
    select * from Production.Product;
    select * from Person.Person;
").ToDataSet<Product, Person>();
```

ExecuteScalar

`ExecuteScalar` methods select a single value. The list of arguments are similar to that of the `Query` methods.

Hide Copy Code

```
object ExecuteScalarSQL(string sql, dynamic param, dynamic outParam,
    SqlTransaction transaction, int? commandTimeout, string connectionString) { }

T ExecuteScalarSQL<T>(string sql, dynamic param, dynamic outParam,
    SqlTransaction transaction, int? commandTimeout, string connectionString) { }
```

The first example returns the largest product id from `Production.Product` table. The second example returns the average length of product names.

Hide Copy Code

```
// get max product id
int maxProductID = SqlHelper.ExecuteScalarSQL<int>("select max(ProductID) from Production.Product");
int maxProductID = SqlHelper.ExecuteScalarSP<int>("sp_get_max_product_id");

// get the average length of product names for products between id 500 and 600
int maxProductID = SqlHelper.ExecuteScalarSQL<int>(@"
    select avg(len(Name))
    from Production.Product
    where ProductID between @FromProductID and @ToProductID
", new { FromProductID = 500, ToProductID = 600 });
```

Execute

`Execute` methods run parameterized SQL and return the number of rows affected. The list of arguments are similar to that of the `Query` methods.

Hide Copy Code

```
int ExecuteSQL(string sql, dynamic param, dynamic outParam,
    SqlTransaction transaction, int? commandTimeout, string connectionString) { }
```

This example executes an update code on the `Production.Product` table and returns a code indicating whether it succe or not.

Hide Copy Code

```
// update specific product name
var outParam = new DynamicParameters
    ("ResultCode", sqlDbType: SqlDbType.Int, direction: ParameterDirection.ReturnValue);

int rowsAffected = SqlHelper.ExecuteSQL(@"
begin try
    begin transaction
        update Production.Product
        set Name = @Name
        where ProductID = @ProductID
    commit transaction
    return 0
end try
begin catch
    rollback transaction
    return -1
end catch
", new { ProductID = 1, Name = "Another Product Name" }, outParam: outParam);

bool succeeded = (outParam.Get<int>("ResultCode") == 0);
```

Working with IEnumerable<dynamic>

There are times when you don't want to work with POCOs. Maybe you need something of the fly. Maybe the database is someone else responsibility and you are not sure what will come back. Maybe the type of result set that will come back is different (different number of columns, different column names, different types), depending on the parameters that you pass. Whatever the reason is,

you are not mapping the results to POCOs. In this case Dapper will return `IEnumerable<dynamic>` to you. The underlying type is really `DapperRow`. It is an internal class that it is not accessible from the outside, meaning you can't cast to `IEnumerable<DapperRow>`. However, you don't need that class to begin with. `dynamic` type can be (and should be) cast to `IDictionary<string, object>` and vice versa. SQLHelper does this for you and more.

In this example, the result set comes back different if you pass a different value to parameter `@Option`. If `@Option = 1` then you get 2 columns of types `string` and `int`. If `@Option = 2` then you get 3 columns of types `datetime`, `decimal` and an undefined type. Creating a POCO for this query will be cumbersome and very ugly code-wise, the POCO will have to include all possible columns. Worst than that, If the different results sets would have shared a column by name but with a different type for each `@Option`, you would need to write several versions of POCOs for each possible value of `@Option`.

Hide Copy Code

```
if @Option = 1
begin
    select StringColumn = 'A', IntColumn = 1 union all
    select StringColumn = 'B', IntColumn = 2
end
else if @Option = 2
begin
    select DateColumn = getdate(),     DecimalColumn = 1.1, NullColumn = null union all
    select DateColumn = getdate() + 1, DecimalColumn = 2.2, NullColumn = null
end
```

ToProperties

SQLHelper provides extension methods `ToProperties` that can take `IEnumerable<dynamic>` and return back `IEnumerable<IDictionary<string, object>>`. If you happened to need it, these extension methods can also take non-dynamic types - `IEnumerable<MyPOCO>` - and still return the appropriate`IEnumerable<IDictionary<string, object>>`. The extension methods can take a list of column names and return just them in the returned results. This can be helpful if you `select *` everything but really only works with just a subset of columns.

Hide Copy Code

```
IEnumerable<IDictionary<string, object>> ToProperties(
    this IEnumerable<object> objs, params string[] columnNames) { }

IEnumerable<IDictionary<string, object>> ToProperties<T>(
    this IEnumerable<T> objs, params string[] columnNames) { }

IEnumerable<IDictionary<string, object>> ToProperties(
    this IEnumerable<IDictionary<string, object>> objs, params string[] columnNames) { }

IDictionary<string, object> ToProperties(
    this IDictionary<string, object> obj, params string[] columnNames) { }

IDictionary<string, object> ToProperties(object obj, params string[] columnNames) { }
```

Following the previous example.

Hide Copy Code

```
string sql = "if @Option = 1 ... ";

IEnumerable<dynamic> dynamicResults = SqlHelper.QuerySQL(sql, new { Option = 1 });
IEnumerable<IDictionary<string, object>> results = dynamicResults.ToProperties();

// one line
var results = SqlHelper.QuerySQL(sql, new { Option = 1 }).ToProperties();
```

ToDataTable

Working with `IEnumerable<IDictionary<string, object>>` might not be the most convenient way of holding data. You might be more comfortable working with a `DataTable`. SQLHelper provides extension method to convert directly from `IEnumerable<dynamic>` to `DataTable`.

Internally, the extension method makes a call to `ToProperties` and retrieves `IEnumerable<IDictionary<string, object>>`. Based on the result set, it determines what the columns (name & type) of the `DataTable` should be, and creates that `DataTable`. Then it adds new rows to the `DataTable` with the appropriate data from the result set. You might notice that constructing this `DataTable` requires 2 iterations over the result set, once to sniff out the columns and once more to populate the `DataTable`. The extension method doesn't make any assumptions about the structure of the various `IDictionary<string, object>`, meaning it doesn't assume they all have the same column names, the same types, the same number of columns. That is the reason why it needs to make 2 passes over the result set. If you are interested only in the structure of the results set, you can set the parameter `toEmptyDataTable = true` and the extension method will not populate the `DataTable`.

Hide Copy Code

```
DataTable ToDataTable(
    this IEnumerable<IDictionary<string, object>> objs,
    bool toEmptyDataTable = false,
    string typeName = null,
    DataTable table = null,
    ValueHandler getValue = null,
    params string[] columnNames
) { }
```

There are a few rules how the `DataTable` is constructed. (1) If the result set is empty then an empty `DataTable` is returned. (2) The default column data type is `object`. So, for a given column, if all the values are `null`, meaning the type can't be determined, then the column data type will default to `object`. (3) If the values are varied in type, then the column data type is set to `object`. For example, for a given column, there are values of type `DateTime` and `decimal`, so the common data type has to be `object`. (4) The `DataTable` columns are a cumulation of all the possible columns from the results set. The structure of the result set rows match each other (column names, types, number of columns) if they all came from a single SQL query. However, if they are disjoint, meaning the rows came from different result sets, for example combining results from two unrelated SQL queries, the extension method will normalize all of them into a single `DataTable`, by combing all the possible columns.

Hide Copy Code

```
IEnumerable<dynamic> dynamicResults = SqlHelper.QuerySQL(
    "select ProductID, Name, ProductNumber from Production.Product"
);
DataTable results = dynamicResults.ToDataTable();

// one line
DataTable products = SqlHelper.QuerySQL(
    "select ProductID, Name, ProductNumber from Production.Product"
).ToDataTable();
```

This example shows how the combination of two result sets come together.

Hide Copy Code

```
// two queries. share only one column ProductID.
IEnumerable<IDictionary<string, object>> products1 = SqlHelper.QuerySQL(
    "select ProductID, Name from Production.Product"
).ToProperties();
IEnumerable<IDictionary<string, object>> products2 = SqlHelper.QuerySQL(
    "select ProductID, ProductNumber from Production.Product"
).ToProperties();

// add them together
List<IDictionary<string, object>> products = new List<IDictionary<string, object>>();
products.AddRange(products1);
products.AddRange(products2);

// get the DataTable
DataTable productsDT = products.ToDataTable();

// three columns
string column0 = productsDT.Columns[0].ColumnName; // ProductID
string column1 = productsDT.Columns[1].ColumnName; // Name
string column2 = productsDT.Columns[2].ColumnName; // ProductNumber
```

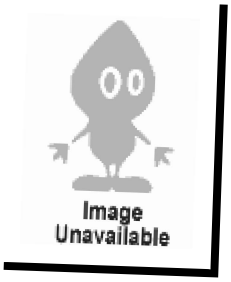
History

- 25/07/2016: Upgraded to Dapper 1.50.2 (.NET 4.5 & .NET Core).
- 28/05/2016: Added support for working with `IEnumerable<dynamic>` (ToProperties & ToDataTable).
- 04/08/2015: Upgraded to Dapper 1.42. This version can handle decimal parameter precision and scale.
- 26/08/2015: Bugfix `QueryMultiple` when returning more that 7 result sets. Now `QueryMultiple` supports up to 14 results sets.
- 10/07/2015: Bugfix `DataTable` to `IEnumerable` value assignment when the table column type and the POCO property type are not the same. Thanks SteveX9.
- 21/12/2015: Minor bugfix `DataTable` to `IEnumerable`.
- 15/01/2016: Bugfix `DataTable` to `IEnumerable` value assignment to POCO property without a setter.

License

Share

About the Author



yuvalsol No Biography provided
Web Developer
Israel

You may also be interested in...

Passing Table Valued Parameters with Dapper

#COBOLrocks TechCasts: New tricks for COBOL devs

Use Dependency Injector "Ninject" to dynamically choose between ORM's (Entity Framework or Dapper) and Databases (SQL Server or Oracle Database)

ES6 Arrow functions - The new fat and concise syntax in JavaScript

Data Mining in C#

Real-Time Communications with Microsoft Edge

Comments and Discussions

Add a Comment or Question

Search Comments Go

First Prev Next

Asp.net core new

Hi yuvalsol,
I hope you are well. I was wondering if you have any plan of upgrading your dapper helper class to the latest version or to work with ASP.NET CORE
Many thanks
Zak

Reply · Email · View Thread · Permalink · Bookmark

Re: Asp.net core new

Done and uploaded. Let me know if you run into any troubles.

Reply · Email · View Thread · Permalink · Bookmark

Re: Asp.net core new

Many thanks

Reply · Email · View Thread · Permalink · Bookmark

Using Async Await Support new

I would like to use dapper but I was wondering if it supports Using Async Await .

Many thanks

Reply · Email · View Thread · Permalink · Bookmark

Re: Using Async Await Support new

Yes, but you're going to need the 4.5 version (this project uses the 4.0 version of Dapper).
There are some good explanations and examples in these links:
Async Dapper with async SQL connection management[^]
Dapper and Dapper Async[^]

Reply · Email · View Thread · Permalink · Bookmark

Need some help to use dapper properly new

i am not familiar with dapper DAL but heard its name and it is popular DAL. how to use dapper for sql server or other database ?

how dapper map data from db to poco classes ?

do we need to write poco classes by hand manually ?

what is the advantage working with dapper ?

From the above article code it seems dapper is just like other Data access layer. i use MSDAAB DAL and it is fine. can u tell me few reason for which people switch to dapper from other DAL.

waiting for your answer . thanks

---

tbhattacharjee

Reply · View Thread · Permalink · Bookmark

Fixed bug new

Hi,
The ToDataTable/ToList is exactly what I was looking for. I did run into an error because I was getting my data from an Excel import. Made these modifications:

Hide Expand Copy Code

```
private static T[] ToEnumerable<T>(DataTable table, Func<T> instanceHandler)
{
    if (table == null)
        return null;

    if (table.Rows.Count == 0)
        return new T[0];

    Type type = typeof(T);

    var columns =
        type.GetFields(BindingFlags.Public | BindingFlags.Instance)
            .Select(f => new
            {
                ColumnName = f.Name,
                IsField = true,
                MemberInfo = (MemberInfo)f,
                propertyType = f.GetType().GetProperty(f.Name).PropertyType  // added
            })
            .Union(
                type.GetProperties(BindingFlags.Public | BindingFlags.Instance)
                    .Where(p => p.CanRead)
                    .Where(p => p.GetGetMethod(true).IsPublic)
                    .Where(p => p.GetIndexParameters().Length == 0)
                    .Select(p => new
                    {
                        ColumnName = p.Name,
                        IsField = false,
                        MemberInfo = (MemberInfo)p,
```

Reply · Email · View Thread · Permalink · Bookmark 5.00/5 (1 vote)

Re: Fixed bug new

Thank you. I fixed the bug and updated the article.
I also took it a little further and added a handler to change the value from one type to another type by the user.
Look for the example under the DataTable To IEnumerable section.

Reply · Email · View Thread · Permalink · Bookmark 5.00/5 (1 vote)

A good start... new

Good article. What you have is a good start to using Dapper, but I feel that Dapper isn't in it's own right an ORM. It is, and admits to being, a Micro-ORM. If you write your own POCO classes and the mappings that go with them, Dapper returns proper POCOs and that's all good but needs significant developer input to write the POCOs and maintain the mappings.

The real power of Dapper comes when you combine it with a code generation tool - something like CodeSmith (v2.6 was public domain, is still available, and does everything you need) or the now defunct MyGeneration, there are others that do the job just as well - Mustache springs to mind.

When you have codegen integrated into your build server that automatically generates your POCOs & mappings by interrogating your database schema, you have a *really* powerful tool. We've been using Dapper that way for a couple of years and it's a godsend. Every time we add or edit a table, view or stored proc the continuous builder server auto-generates the updated POCOs & mappings. Hassle-free, very fast, and has saved us thousands of lines of tedious ADO hand-coding. Sam Saffron, if on the off-chance you read this, thanks for Dapper...

modified 25-Mar-15 18:58pm.

Reply · View Thread · Permalink · Bookmark 5.00/5 (2 votes)

Re: A good start... new

I never use Dapper. just heard it is micro ORM. what is micro ORM ?

do we need to write POCO classes by hand when working with dapper ?

here you brief that you use some tool which generate or Update POCO classes. is it code smith ?
code smith is free tool and how to generate poco with code smith.

from the above article code it seems dapper is just like other Data access layer. i use MSDAAB and it is fine. can u tell me few reason for which people switch to dapper from other DAL.

waiting for your answer . thanks

tbhattacharjee

Reply · View Thread · Permalink · Bookmark

Last Visit: 31-Dec-99 23:00 Last Update: 25-Jul-16 17:43 Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin