E-mail
(mailto:j%72@jr%63oo%6b.%6
@Eldorian
(http://twitter.com/eldorian)
LinkedIn
(https://www.linkedin.com/in/jrcc
GitHub
(https://github.com/Eldorian)

ABOUT

About Me
(https://jrcook.dev/about-me/)

Search for:

RECENT POSTS

My Top Video Games of 2017
(https://jrcook.dev/2018/01/03/my-top-video-games-of-2017/)

How to Get Bluetooth Headphones to Work with Nintendo Switch
(https://jrcook.dev/2017/06/22/how-to-get-bluetooth-headphones-to-work-with-nintendo-switch/)

My Top Video Games of 2016
(https://jrcook.dev/2017/01/03/my-top-video-games-of-2016/)

Making the World Connect Again
(https://jrcook.dev/2016/11/11/making-the-world-connect-again/)

Using Dapper with DB2
(https://jrcook.dev/2016/10/24/using-dapper-with-db2/)

META

Log in (https://jrcook.dev/wp-login.php)

Entries feed
(https://jrcook.dev/feed/)

Comments feed
(https://jrcook.dev/comments/feed/)

WordPress.org
(https://wordpress.org/)

TAG CLOUD

.NET (https://jrcook.dev/tag/net/)

API (https://jrcook.dev/tag/api/)

C# (https://jrcook.dev/tag/c/)

Using Dapper with DB2 (https://jrcook.dev/2016/10/24/using-dapper-with-db2/)

OCT 24, 2016 0 COMMENTS (HTTPS://JRCOOK.DEV/2016/10/24/USING-DAPPER-WITH-DB2/#DISQUS_THREAD)

I know the new hotness is development is always looking for Greenfield projects and building out your new applications using the new technology and building out your database using Code First with Entity Framework.

However, life in the development world isn't all green fields and puppies and you're finding yourself having to use legacy systems for one reason or another. I've found myself in that position in needing to interact with a legacy DB2 database and create a Web API in order to create some services that can produce and consume that data.

In the past we've used a powerful ORM tool like Entity Framework to do our CRUD operations to the DB2 database. We definitely had a lot of pain points in using EF, most importantly speed and performance being the culprit. It was so bad that there were times we ended up just resorting to ADO.NET. Yes, it was messier code and tougher to unit test, but it was also incredibly faster.

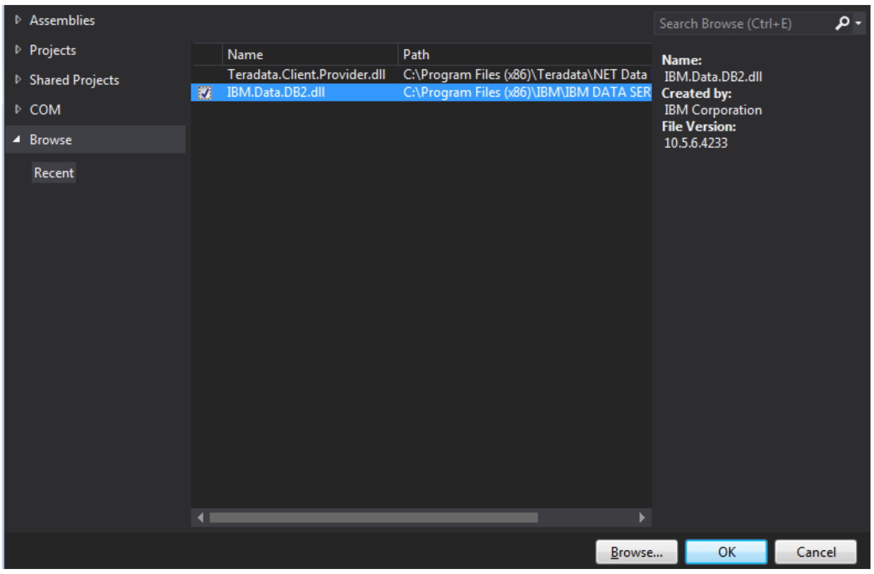
Due to this, on a Web API I was creating, I decided to do some research on other ORMs and came across the Micro ORM that Stack Overflow created called Dapper (https://github.com/StackExchange/dapper-dot-net). Dapper promised speed close to ADO.NET while also providing extensions for your IDbConnection interface. Upon using this I was blown away by the speed and also how simple it was to set up.

Let me walk you through a quick and simple repository and in some future blogs I will show how I implemented the Web API portion using other nuget packages like Automapper and Fluentvalidation.

The first thing you will need to do is download and install the ODBC and CLI client DB2 drivers from IBM. More information on that here (https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwjQy
01.ibm.com%2Fsupport%2Fdocview.wss%3Fuid%3Dswg21418043&usg=AFQjCN
n-zczJ7deQf_fIG7G-Lbg).

Dapper (https://jrcook.dev/tag/dapper/)
(https://jrcook.dev/tag/db2/)
Nintendo Switch (https://jrcook.dev/tag/nintendo-switch/)
PC Gaming (https://jrcook.dev/tag/pc-gaming/)
Playstation 4 (https://jrcook.dev/tag/playstation-4/)

In your newly created .NET project you will then add a reference and browse to the location of IBM.Data.DB2.dll and add that as a reference. This is what will allow you to now use DB2Connection extensions. The DLL is most commonly found in this location if you did a non custom install for 64 bit or (x86) for 32 bit—C:\Program Files\IBM\IBM DATA SERVER DRIVER\bin\netf40



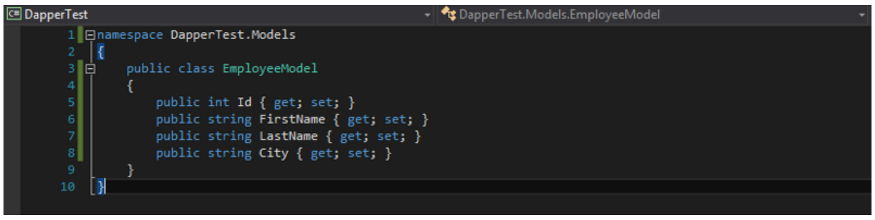
Name	Path
Teradata.Client.Provider.dll	C:\Program Files (x86)\Teradata\NET Data
IBM.Data.DB2.dll	C:\Program Files (x86)\IBM\IBM DATA SER

Name:
IBM.Data.DB2.dll
Created by:
IBM Corporation
File Version:
10.5.6.4233

The next thing you will want to do is add the Dapper reference using Nuget Package Manager. This is quite simple to do. Just go to your Package Manager Console and type in the following: **Install-Package Dapper**

This should quickly install Dapper as a reference to your project and you are ready to begin coding. Now I don't want to get into dependency injection or unit testing with this example—but you should totally use it and fully unit test your code, those examples will be for another day.

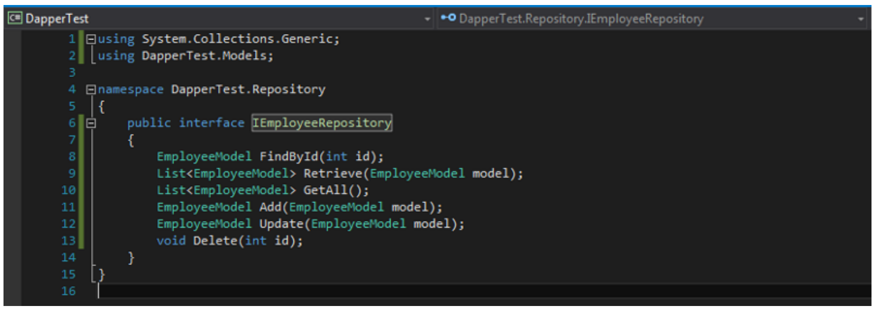
The first thing I want to do is create my model. This is how my database table will look like that I want to do CRUD operations with. For example purposes I made a very simple EmployeeModel that has an id, First Name, Last Name, and City.



```
namespace DapperTest.Models
{
    public class EmployeeModel
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string City { get; set; }
    }
}
```

The EmployeeModel

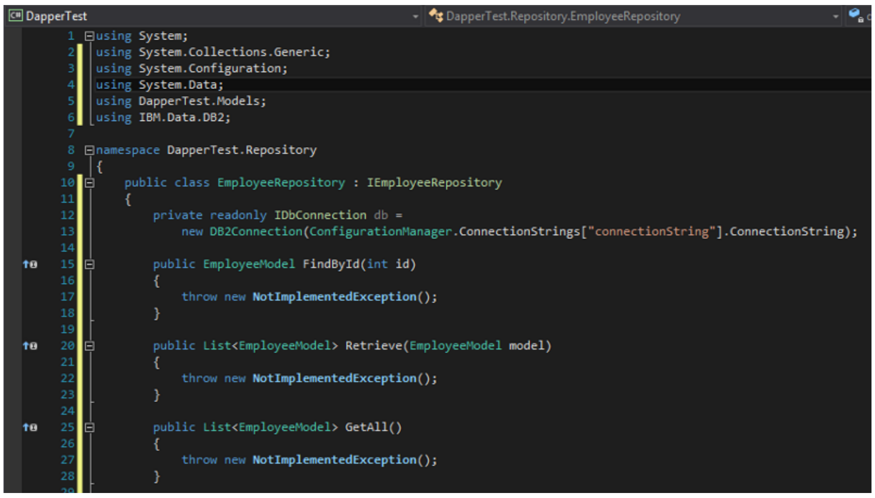
The second thing I do is create a very simple Interface for my repository. I want to be able to Find by Id, Get all records, Retrieve, Add, Update, and Delete. I'm going to show Find by Id, Get All, Add, and Delete in this example. For Retrieve and Update I use something called Dapper.SqlBuilder and I'd like to discuss that alone in a future blog.



```
using System.Collections.Generic;
using DapperTest.Models;

namespace DapperTest.Repository
{
    public interface IEmployeeRepository
    {
        EmployeeModel FindById(int id);
        List<EmployeeModel> Retrieve(EmployeeModel model);
        List<EmployeeModel> GetAll();
        EmployeeModel Add(EmployeeModel model);
        EmployeeModel Update(EmployeeModel model);
        void Delete(int id);
    }
}
```

My Interface I will be using for EmployeeRepository

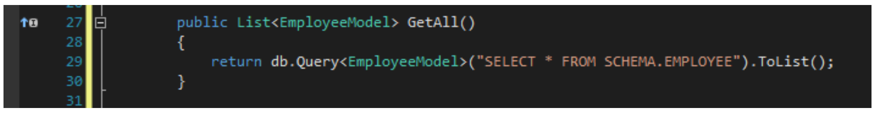
And finally I will create an EmployeeRepository class that will implement the interface I just created. I will also go ahead and set up my IDbConnection using DB2Connection and reference my connection string that I have in my web.config. This is where normally you would want to start using Dependency Injection so you can create mocks of your code for unit testing purposes—but for ease of setting up I will by pass that.



```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using DapperTest.Models;
using IBM.Data.DB2;

namespace DapperTest.Repository
{
    public class EmployeeRepository : IEmployeeRepository
    {
        private readonly IDbConnection db =
            new DB2Connection(ConfigurationManager.ConnectionStrings["connectionString"].ConnectionString);

        public EmployeeModel FindById(int id)
        {
            throw new NotImplementedException();
        }

        public List<EmployeeModel> Retrieve(EmployeeModel model)
        {
            throw new NotImplementedException();
        }

        public List<EmployeeModel> GetAll()
        {
            throw new NotImplementedException();
        }
```

A blank repo implementing my interface

The first method we'll do is GetAll, it's really simple. All we want to do is run a Query using our EmployeeModel, return all records in a list. It's a very simple SQL statement.



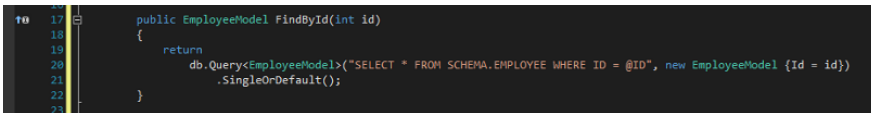
```
        public List<EmployeeModel> GetAll()
        {
            return db.Query<EmployeeModel>("SELECT * FROM SCHEMA.EMPLOYEE").ToList();
        }
```

This will return a list of EmployeeModel from DB2

The second method we will implement is FindById. In this case we are passing in an integer and we want to return 1 record from our

Employee table. You'll notice the SQL in this case I am using @ID as it is very important that you parameter your SQL to defend against injection attacks. In this case it knows that id is what is being passed on, I'm using it as a parameter and using an anonymous object to assign my EmployeeModel Id to the variable id being passed in.

This will then return the record in the database where the Id equals what we passed in.

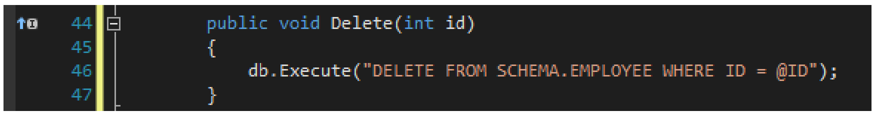


```
        public EmployeeModel FindById(int id)
        {
            return
                db.Query<EmployeeModel>("SELECT * FROM SCHEMA.EMPLOYEE WHERE ID = @ID", new EmployeeModel {Id = id})
                    .SingleOrDefault();
        }
```

Always use parameters in your SQL!

The third method I want to show is delete. It's also very simple and works much the same way we saw in the last 2 examples. In real life I hardly ever use delete as a method in my APIs and would likely want to return something if I did, like a message or something to that effect. However, for example sake I went with a very easy void method.

The main difference is we don't have to do a query here, we can just straight up execute our SQL. Remember to use a parameter!



```
        public void Delete(int id)
        {
            db.Execute("DELETE FROM SCHEMA.EMPLOYEE WHERE ID = @ID");
        }
```

And for the final method I want to show today is our Add method. This one gets to be a little more complex because what I want to do is add the record, and then return the record I just created back.

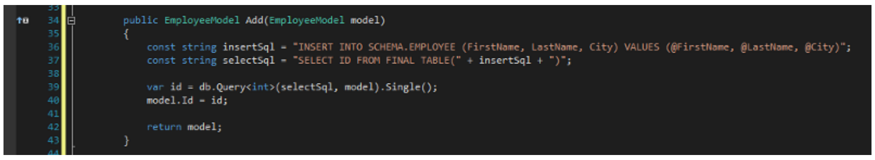
Remember that in this case Id is an autonumber that is created by the database when a record is created. It's a unique identifier that will not be passed in through the method. With SQL Server you can just use @SCOPE_IDENTITY to accomplish this but DB2 is a bit more rusty with how to do it.

Instead we want to select the Id from the FINAL TABLE of the INSERT statement we do. Once we get the id that was just created we can run a query on that table, add it back to the model we passed in, and return it back.

Seriously, good luck finding any other place on the internet that describes this, I tried. One of our DBAs was able to figure out a method for me that worked without making a second trip to the database.

NOTE: There is something in DB2 called "IDENTITY_VAL_LOCAL" but I'm not sure if that solely works with DB2 for

Linux/Unix/Windows – I was unable to get it to work with Z/OS.



```
public EmployeeModel Add(EmployeeModel model)
{
    const string insertSql = "INSERT INTO SCHEMA.EMPLOYEE (FirstName, LastName, City) VALUES (@FirstName, @LastName, @City)";
    const string selectSql = "SELECT ID FROM FINAL TABLE(" + insertSql + ")";

    var id = db.Query<int>(selectSql, model).Single();
    model.Id = id;

    return model;
}
```

And that should do it! We still have Retrieve and Update to accomplish but those will be for my next blog as I explain how to use Dapper.SqlBuilder for those. After that I will show you how to wrap this all up in a Web API.

This is my first ever post I have made about programming, so if you liked it please let me know. It won't be perfect so if you have any suggestions for improvements I would love to hear that as well. You can do so here or hit me up on Twitter (http://twitter.com/Eldorian).

Good luck and happy coding!

.NET (https://jrcook.dev/tag/net/) *API (https://jrcook.dev/tag/api/)* *C# (https://jrcook.dev/tag/c/)*
Dapper (https://jrcook.dev/tag/dapper/) *DB2 (https://jrcook.dev/tag/db2/)*