eBook Gratuit

APPRENEZ Dapper.NET

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#dapper

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: dapper-net

It is an unofficial and free Dapper.NET ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Dapper.NET.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Dapper.NET

Remarques

Qu'est ce que Dapper?

Dapper est un micro-ORM pour .Net qui étend votre `IDbConnection` , simplifiant la configuration, l'exécution et la lecture des résultats.

Comment puis-je l'obtenir?

- github: https://github.com/StackExchange/dapper-dot-net
- NuGet: https://www.nuget.org/packages/Dapper

Tâches communes

- Requête de base
- Exécution des commandes

Versions

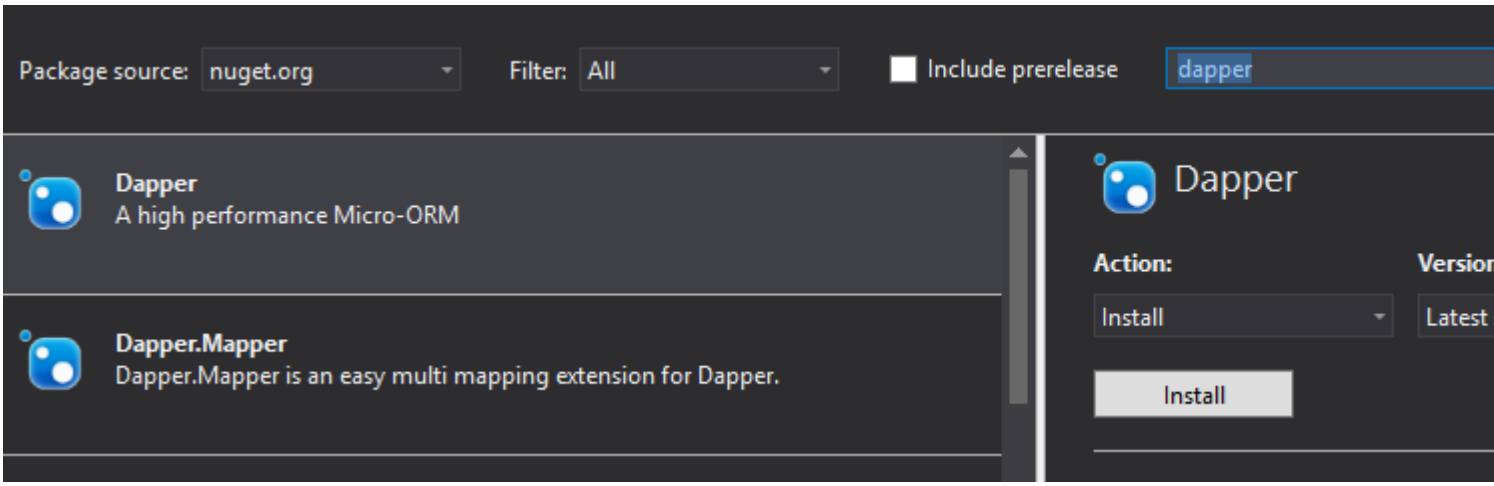
Version	Remarques	Date de sortie
1,50.0	core-clr / asp.net 5.0 contre RTM	2016-06-29
1,42.0		2015-05-06
1,40.0		2015-04-03
1,30.0		2014-08-14
1.20.0		2014-05-08
1.10.0		2012-06-27
1.0.0		2011-04-14

Examples

Installer Dapper à partir de Nuget

Soit la recherche dans l'interface graphique de Visual Studio:

Outils> Gestionnaire de packages NuGet> Gérer les packages pour solution ... (Visual Studio 2015)



Ou exécutez cette commande dans une instance Nuget Power Shell pour installer la dernière version stable

```
Install-Package Dapper
```

Ou pour une version spécifique

```
Install-Package Dapper -Version 1.42.0
```

Utiliser Dapper en C

```
using System.Data;
using System.Linq;
using Dapper;

class Program
{
    static void Main()
    {
        using (IDbConnection db = new
SqlConnection("Server=myServer;Trusted_Connection=true"))
        {
            db.Open();
            var result = db.Query<string>("SELECT 'Hello World'").Single();
            Console.WriteLine(result);
        }
    }
}
```

Envelopper la connexion dans un bloc `Using` va fermer la connexion

Utiliser Dapper dans LINQPad

LINQPad est idéal pour tester les requêtes de base de données et inclut l' intégration de NuGet . Pour utiliser Dapper dans LINQPad, appuyez sur **F4** pour ouvrir les propriétés de la requête, puis

sélectionnez **Ajouter NuGet** . Recherchez **dapper dot net** et sélectionnez **Ajouter à la requête** . Vous souhaiterez également cliquer sur **Ajouter des espaces de noms** et mettre en surbrillance Dapper pour inclure les méthodes d'extension dans votre requête LINQPad.

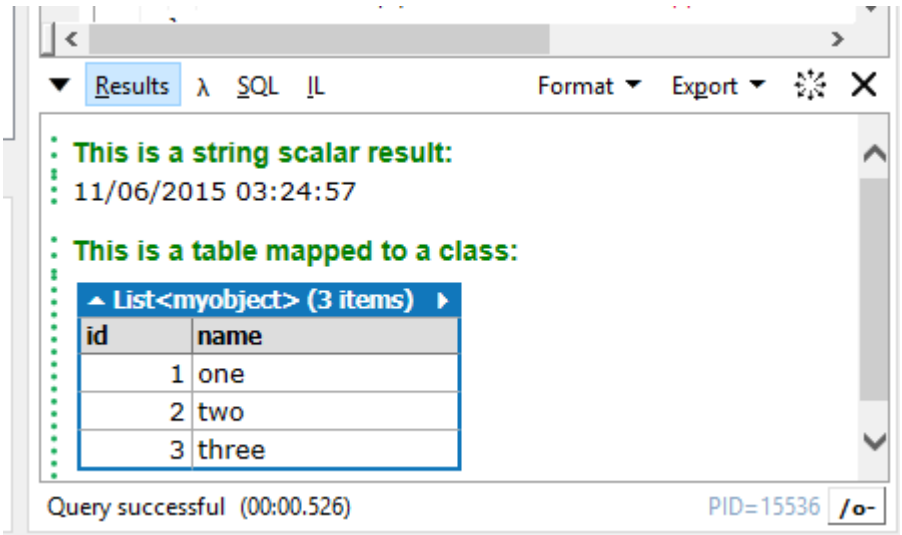
Une fois que Dapper est activé, vous pouvez modifier la liste déroulante Langue en **Programme** C #, associer les résultats de la requête aux classes C # et utiliser la méthode .Dump () pour inspecter les résultats:

```
void Main()
{
    using (IDbConnection db = new SqlConnection("Server=myServer;Trusted_Connection=true")){
        db.Open();
        var scalar = db.Query<string>("SELECT GETDATE()").SingleOrDefault();
        scalar.Dump("This is a string scalar result:");

        var results = db.Query<myobject>(@"
        SELECT * FROM (
        VALUES (1,'one'),
            (2,'two'),
            (3,'three')
        ) AS mytable(id,name)");
        results.Dump("This is a table mapped to a class:");
    }
}

// Define other methods and classes here
class myobject {
    public int id { get; set; }
    public string name { get; set; }
}
```

Les résultats lors de l'exécution du programme ressembleraient à ceci:



Lire Démarrer avec Dapper.NET en ligne: https://riptutorial.com/fr/dapper/topic/2/demarrer-avec-dapper-net

Chapitre 2: Exécution des commandes

Examples

Exécuter une commande qui ne renvoie aucun résultat

```
IDBConnection db = /* ... */
var id = /* ... */

db.Execute(@"update dbo.Dogs set Name = 'Beowoof' where Id = @id",
   new { id });
```

Procédures stockées

Usage simple

Dapper prend entièrement en charge les processus stockés:

```
var user = conn.Query<User>("spGetUser", new { Id = 1 },
                            commandType: CommandType.StoredProcedure)
           .SingleOrDefault();
```

Paramètres d'entrée, de sortie et de retour

Si vous voulez quelque chose de plus chic, vous pouvez faire:

```
var p = new DynamicParameters();
p.Add("@a", 11);
p.Add("@b",
      dbType: DbType.Int32,
      direction: ParameterDirection.Output);
p.Add("@c",
      dbType: DbType.Int32,
      direction: ParameterDirection.ReturnValue);

conn.Execute("spMagicProc", p,
             commandType: CommandType.StoredProcedure);

var b = p.Get<int>("@b");
var c = p.Get<int>("@c");
```

Paramètres de la table

Si vous avez une procédure stockée qui accepte un paramètre de valeur de table, vous devez transmettre un DataTable qui a la même structure que le type de tableau dans SQL Server. Voici une définition pour un type de tableau et une procédure l'utilisant:

```
CREATE TYPE [dbo].[myUDTT] AS TABLE([i1] [int] NOT NULL);
GO
CREATE PROCEDURE myProc(@data dbo.myUDTT readonly) AS
SELECT i1 FROM @data;
GO
/*
-- optionally grant permissions as needed, depending on the user you execute this with.
-- Especially the GRANT EXECUTE ON TYPE is often overlooked and can cause problems if omitted.
GRANT EXECUTE ON TYPE::[dbo].[myUDTT] TO [user];
GRANT EXECUTE ON dbo.myProc TO [user];
GO
*/
```

Pour appeler cette procédure depuis c #, vous devez procéder comme suit:

```
// Build a DataTable with one int column
DataTable data = new DataTable();
data.Columns.Add("i1", typeof(int));
// Add two rows
data.Rows.Add(1);
data.Rows.Add(2);

var q = conn.Query("myProc", new {data}, commandType: CommandType.StoredProcedure);
```

Lire Exécution des commandes en ligne: https://riptutorial.com/fr/dapper/topic/5/execution-des-commandes

Chapitre 3: Gestionnaires de types

Remarques

Les gestionnaires de type permettent aux types de base de données d'être convertis en types personnalisés .Net.

Examples

Conversion de varchar en IHtmlString

```
public class IHtmlStringTypeHandler : SqlMapper.TypeHandler<IHtmlString>
{
    public override void SetValue(
        IDbDataParameter parameter,
        IHtmlString value)
    {
        parameter.DbType = DbType.String;
        parameter.Value = value?.ToHtmlString();
    }

    public override IHtmlString Parse(object value)
    {
        return MvcHtmlString.Create(value?.ToString());
    }
}
```

Installation d'un TypeHandler

Le gestionnaire de type ci-dessus peut être installé dans `SqlMapper` à l'aide de la méthode `AddTypeHandler` .

```
SqlMapper.AddTypeHandler<IHtmlString>(new IHtmlStringTypeHandler());
```

L'inférence de type vous permet d'omettre le paramètre de type générique:

```
SqlMapper.AddTypeHandler(new IHtmlStringTypeHandler());
```

Il y a aussi une surcharge à deux arguments qui prend un argument de `Type` explicite:

```
SqlMapper.AddTypeHandler(typeof(IHtmlString), new IHtmlStringTypeHandler());
```

Lire Gestionnaires de types en ligne: https://riptutorial.com/fr/dapper/topic/6/gestionnaires-de-types

Chapitre 4: Inserts en vrac

Remarques

`WriteToServer` et `WriteToServerAsync` ont des surcharges qui acceptent les tableaux IDataReader (vus dans les exemples), DataTable et DataRow (`DataRow[]`) comme source des données pour la copie en bloc.

Examples

Async Bulk Copy

Cet exemple utilise une méthode `ToDataReader` décrite ici Création d'un DataReader de liste générique pour SqlBulkCopy .

Cela peut également être fait en utilisant des méthodes non asynchrones.

```
public class Widget
{
    public int WidgetId {get;set;}
    public string Name {get;set;}
    public int Quantity {get;set;}
}

public async Task AddWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        await conn.OpenAsync();

        using(var bulkCopy = new SqlBulkCopy(conn)) {
            bulkCopy.BulkCopyTimeout = SqlTimeoutSeconds;
            bulkCopy.BatchSize = 500;
            bulkCopy.DestinationTableName = "Widgets";
            bulkCopy.EnableStreaming = true;

            using(var dataReader = widgets.ToDataReader())
            {
                await bulkCopy.WriteToServerAsync(dataReader);
            }
        }
    }
}
```

Copie en vrac

Cet exemple utilise une méthode `ToDataReader` décrite ici Création d'un DataReader de liste générique pour SqlBulkCopy .

Cela peut également être fait en utilisant des méthodes asynchrones.

```
public class Widget
{
    public int WidgetId {get;set;}
    public string Name {get;set;}
    public int Quantity {get;set;}
}

public void AddWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        conn.Open();

        using(var bulkCopy = new SqlBulkCopy(conn)) {
            bulkCopy.BulkCopyTimeout = SqlTimeoutSeconds;
            bulkCopy.BatchSize = 500;
            bulkCopy.DestinationTableName = "Widgets";
            bulkCopy.EnableStreaming = true;

            using(var dataReader = widgets.ToDataReader())
            {
                bulkCopy.WriteToServer(dataReader);
            }
        }
    }
}
```

Lire Inserts en vrac en ligne: https://riptutorial.com/fr/dapper/topic/6279/inserts-en-vrac

Chapitre 5: Manipulation des Nulls

Examples

null vs DBNull

Dans ADO.NET, la gestion correcte de `null` est une source constante de confusion. Le point clé de Dapper est que *vous n'avez pas à le faire* ; il traite de tout en interne.

- les valeurs de paramètre qui sont `null` sont correctement envoyées en tant que `DBNull.Value`
- les valeurs lues qui sont `null` sont présentées comme `null` ou (dans le cas d'un mappage sur un type connu) simplement ignorées (laissant leur type par défaut)

Ça marche:

```
string name = null;
int id = 123;
connection.Execute("update Customer set Name=@name where Id=@id",
    new {id, name});
```

Lire Manipulation des Nulls en ligne: https://riptutorial.com/fr/dapper/topic/13/manipulation-des-nulls

Chapitre 6: Multimapping

Syntaxe

- `public static IEnumerable<TReturn> Query<TFirst, TSecond, TReturn>(this IDbConnection cnn, string sql, Func<TFirst, TSecond, TReturn> map, object param = null, IDbTransaction transaction = null, bool buffered = true, string splitOn = "Id", int? commandTimeout = null, CommandType? commandType = null)`
- `public static IEnumerable<TReturn> Query<TFirst, TSecond, TThird, TFourth, TFifth, TSixth, TSeventh, TReturn>(this IDbConnection cnn, string sql, Func<TFirst, TSecond, TThird, TFourth, TFifth, TSixth, TSeventh, TReturn> map, object param = null, IDbTransaction transaction = null, bool buffered = true, string splitOn = "Id", int? commandTimeout = null, CommandType? commandType = null)`
- `public static IEnumerable<TReturn> Query<TReturn>(this IDbConnection cnn, string sql, Type[] types, Func<object[], TReturn> map, object param = null, IDbTransaction transaction = null, bool buffered = true, string splitOn = "Id", int? commandTimeout = null, CommandType? commandType = null)`

Paramètres

Paramètre	Détails
CNN	Votre connexion à la base de données, qui doit déjà être ouverte.
sql	Commande à exécuter.
les types	Tableau de types dans le jeu d'enregistrements.
carte	`Func<>` qui gère la construction du résultat de retour.
param	Objet pour extraire les paramètres de.
transaction	Transaction dont cette requête fait partie, le cas échéant.
tamponné	S'il faut ou non mettre en mémoire tampon les résultats de la requête. Ceci est un paramètre facultatif avec la valeur par défaut étant true. Lorsque la mise en mémoire tampon est vraie, les résultats sont mis en mémoire tampon dans une `List<T>` , puis renvoyés sous la forme d'un `IEnumerable<T>` sûr pour une énumération multiple. Lorsque la mise en mémoire tampon est fausse, la connexion SQL est maintenue ouverte jusqu'à ce que vous ayez fini de lire, ce qui vous permet de traiter une seule ligne à la fois en mémoire. Plusieurs énumérations engendreront des connexions supplémentaires à la base de données. Bien que false mis en mémoire tampon soit très efficace pour réduire l'utilisation de la mémoire si vous ne gérez que de très petits fragments d'enregistrements renvoyés, il se caractérise par une surcharge de performances considérable par rapport à la matérialisation rapide du jeu de résultats. Enfin, si vous avez de nombreuses connexions SQL non tamponnées simultanées, vous devez tenir compte de la famine du

Paramètre	Détails
	pool de connexions, ce qui entraîne le blocage des requêtes jusqu'à ce que les connexions soient disponibles.
splitOn	Le champ que nous devons diviser et lire le second objet (par défaut: id). Cela peut être une liste délimitée par des virgules lorsque plus d'un type est contenu dans un enregistrement.
commandeTimeout	Nombre de secondes avant l'expiration du délai d'exécution de la commande.
type de commande	Est-ce un processus stocké ou un lot?

Examples

Mappage multi-tables simple

Disons que nous avons une interrogation des cavaliers restants qui doivent remplir une classe de personnes.

prénom	Née	Résidence
Daniel Dennett	1942	les États-Unis d'Amérique
Sam Harris	1967	les États-Unis d'Amérique
Richard dawkins	1941	Royaume-Uni

```
public class Person
{
    public string Name { get; set; }
    public int Born { get; set; }
    public Country Residience { get; set; }
}

public class Country
{
    public string Residence { get; set; }
}
```

Nous pouvons remplir la classe de personne ainsi que la propriété Residence avec une instance de Country à l'aide d'une `Query<>` surcharge `Query<>` qui prend un `Func<>` pouvant être utilisé pour composer l'instance renvoyée. Le `Func<>` peut prendre jusqu'à 7 types d'entrées, l'argument générique final étant toujours le type de retour.

```
var sql = @"SELECT 'Daniel Dennett' AS Name, 1942 AS Born, 'United States of America' AS
Residence
UNION ALL SELECT 'Sam Harris' AS Name, 1967 AS Born, 'United States of America' AS Residence
UNION ALL SELECT 'Richard Dawkins' AS Name, 1941 AS Born, 'United Kingdom' AS Residence";
```

```
var result = connection.Query<Person, Country, Person>(sql, (person, country) => {
        if(country == null)
        {
            country = new Country { Residence = "" };
        }
        person.Residience = country;
        return person;
    },
    splitOn: "Residence");
```

Notez l'utilisation de l' `splitOn: "Residence"` qui est la 1ère colonne du prochain type de classe à renseigner (dans ce cas, `Country`). Dapper recherchera automatiquement une colonne appelée *Id* à diviser mais si elle n'en trouve pas et que `splitOn` n'est pas fourni, une `System.ArgumentException` sera lancée avec un message utile. Donc, bien que ce soit facultatif, vous devrez généralement fournir une valeur `splitOn` .

Cartographie un à plusieurs

Regardons un exemple plus complexe qui contient une relation un-à-plusieurs. Notre requête contiendra désormais plusieurs lignes contenant des données en double et nous devrons gérer cela. Nous faisons cela avec une recherche dans une fermeture.

La requête change légèrement comme le font les exemples de classes.

Id	prénom	Née	CountryId	Nom du pays	BookId	Nom du livre
1	Daniel Dennett	1942	1	les États-Unis d'Amérique	1	Brainstorms
1	Daniel Dennett	1942	1	les États-Unis d'Amérique	2	Espace vital
2	Sam Harris	1967	1	les États-Unis d'Amérique	3	Le paysage moral
2	Sam Harris	1967	1	les États-Unis d'Amérique	4	Se réveiller: Guide de spiritualité sans religion
3	Richard dawkins	1941	2	Royaume-Uni	5	La magie de la réalité: comment nous savons ce qui est vraiment vrai
3	Richard dawkins	1941	2	Royaume-Uni	6	Un appétit pour merveille: la fabrication d'un scientifique

```
public class Person
{
    public int Id { get; set; }
```

```
    public string Name { get; set; }
    public int Born { get; set; }
    public Country Residience { get; set; }
    public ICollection<Book> Books { get; set; }
}

public class Country
{
    public int CountryId { get; set; }
    public string CountryName { get; set; }
}

public class Book
{
    public int BookId { get; set; }
    public string BookName { get; set; }
}
```

Les dictionnaires `remainingHorsemen` seront remplis d'instances entièrement matérialisées des objets de la personne. Pour chaque ligne du résultat de la requête, les valeurs mappées des instances des types définis dans les arguments lambda sont transmises et il vous appartient de gérer cela.

```
            var sql = @"SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS
CountryId, 'United States of America' AS CountryName, 1 AS BookId, 'Brainstorms' AS BookName
UNION ALL SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS CountryId, 'United
States of America' AS CountryName, 2 AS BookId, 'Elbow Room' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States
of America' AS CountryName, 3 AS BookId, 'The Moral Landscape' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States
of America' AS CountryName, 4 AS BookId, 'Waking Up: A Guide to Spirituality Without Religion'
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United
Kingdom' AS CountryName, 5 AS BookId, 'The Magic of Reality: How We Know What`s Really True'
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United
Kingdom' AS CountryName, 6 AS BookId, 'An Appetite for Wonder: The Making of a Scientist' AS
BookName";

var remainingHorsemen = new Dictionary<int, Person>();
connection.Query<Person, Country, Book, Person>(sql, (person, country, book) => {
    //person
    Person personEntity;
    //trip
    if (!remainingHorsemen.TryGetValue(person.Id, out personEntity))
    {
        remainingHorsemen.Add(person.Id, personEntity = person);
    }

    //country
    if(personEntity.Residience == null)
    {
        if (country == null)
        {
            country = new Country { CountryName = "" };
        }
        personEntity.Residience = country;
    }
```

```
    //books
    if(personEntity.Books == null)
    {
        personEntity.Books = new List<Book>();
    }

    if (book != null)
    {
        if (!personEntity.Books.Any(x => x.BookId == book.BookId))
        {
            personEntity.Books.Add(book);
        }
    }

    return personEntity;
},
splitOn: "CountryId,BookId");
```

Notez comment l'argument `splitOn` est une liste délimitée par des virgules des premières colonnes du type suivant.

Cartographie de plus de 7 types

Parfois, le nombre de types que vous mappez dépasse les 7 fournis par le Func <> qui effectue la construction.

Au lieu d'utiliser la `Query<>` avec les entrées d'argument de type générique, nous allons fournir les types à mapper en tant que tableau, suivis de la fonction de mappage. Outre le réglage manuel initial et la conversion des valeurs, le reste de la fonction ne change pas.

```
            var sql = @"SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS
CountryId, 'United States of America' AS CountryName, 1 AS BookId, 'Brainstorms' AS BookName
UNION ALL SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS CountryId, 'United
States of America' AS CountryName, 2 AS BookId, 'Elbow Room' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States
of America' AS CountryName, 3 AS BookId, 'The Moral Landscape' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States
of America' AS CountryName, 4 AS BookId, 'Waking Up: A Guide to Spirituality Without Religion'
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United
Kingdom' AS CountryName, 5 AS BookId, 'The Magic of Reality: How We Know What`s Really True'
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United
Kingdom' AS CountryName, 6 AS BookId, 'An Appetite for Wonder: The Making of a Scientist' AS
BookName";

var remainingHorsemen = new Dictionary<int, Person>();
connection.Query<Person>(sql,
    new[]
    {
        typeof(Person),
        typeof(Country),
        typeof(Book)
    }
    , obj => {

        Person person = obj[0] as Person;
```

```
        Country country = obj[1] as Country;
        Book book = obj[2] as Book;

        //person
        Person personEntity;
        //trip
        if (!remainingHorsemen.TryGetValue(person.Id, out personEntity))
        {
            remainingHorsemen.Add(person.Id, personEntity = person);
        }

        //country
        if(personEntity.Residience == null)
        {
            if (country == null)
            {
                country = new Country { CountryName = "" };
            }
            personEntity.Residience = country;
        }

        //books
        if(personEntity.Books == null)
        {
            personEntity.Books = new List<Book>();
        }

        if (book != null)
        {
            if (!personEntity.Books.Any(x => x.BookId == book.BookId))
            {
                personEntity.Books.Add(book);
            }
        }

        return personEntity;
},
splitOn: "CountryId,BookId");
```

Mappages personnalisés

Si les noms des colonnes de requête ne correspondent pas à vos classes, vous pouvez configurer des mappages pour les types. Cet exemple illustre le mappage à l'aide de `System.Data.Linq.Mapping.ColumnAttribute` ainsi qu'un mappage personnalisé.

> Les mappages ne doivent être configurés qu'une seule fois par type, donc définissez-les au démarrage de l'application ou ailleurs pour qu'ils ne soient initialisés qu'une seule fois.

En supposant la même requête que l'exemple One-to-many et les classes remaniées vers de meilleurs noms comme ceux-ci:

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Born { get; set; }
```

```
    public Country Residience { get; set; }
    public ICollection<Book> Books { get; set; }
}

public class Country
{
    [System.Data.Linq.Mapping.Column(Name = "CountryId")]
    public int Id { get; set; }

    [System.Data.Linq.Mapping.Column(Name = "CountryName")]
    public string Name { get; set; }
}

public class Book
{
    public int Id { get; set; }

    public string Name { get; set; }
}
```

> Notez que `Book` ne repose pas sur `ColumnAttribute` mais que nous devons conserver l'instruction `if`

Placez maintenant ce code de mappage quelque part dans votre application où il n'est exécuté qu'une seule fois:

```
Dapper.SqlMapper.SetTypeMap(
    typeof(Country),
    new CustomPropertyTypeMap(
        typeof(Country),
        (type, columnName) =>
            type.GetProperties().FirstOrDefault(prop =>
                prop.GetCustomAttributes(false)
                    .OfType<System.Data.Linq.Mapping.ColumnAttribute>()
                    .Any(attr => attr.Name == columnName)))
);


var bookMap = new CustomPropertyTypeMap(
    typeof(Book),
    (type, columnName) =>
    {
        if(columnName == "BookId")
        {
            return type.GetProperty("Id");
        }

        if (columnName == "BookName")
        {
            return type.GetProperty("Name");
        }

        throw new InvalidOperationException($"No matching mapping for {columnName}");
    }
);
Dapper.SqlMapper.SetTypeMap(typeof(Book), bookMap);
```

La requête est ensuite exécutée à l'aide de l'un des exemples de `Query<>` précédents `Query<>` .

Une manière plus simple d'ajouter les mappages est montrée dans cette réponse .

Lire Multimapping en ligne: https://riptutorial.com/fr/dapper/topic/351/multimapping

Chapitre 7: Paramètre Syntaxe Référence

Paramètres

Paramètre	Détails
`this cnn`	La connexion à la base de données sous-jacente - `this` indique une méthode d'extension; la connexion n'a pas besoin d'être ouverte - si elle n'est pas ouverte, elle est ouverte et fermée automatiquement.
`<T>` / `Type`	(facultatif) Type d'objet à renvoyer; Si l'API non-générique / non- `Type` est utilisée, un objet `dynamic` est renvoyé par ligne, simulant une propriété nommée par nom de colonne renvoyée par la requête (cet objet `dynamic` implémente également `IDicionary<string,object>`).
`sql`	Le SQL à exécuter
`param`	(facultatif) Les paramètres à inclure.
`transaction`	(facultatif) La transaction de base de données à associer à la commande
`buffered`	(facultatif) Indique s'il faut pré-consommer les données dans une liste (valeur par défaut) plutôt que d'exposer un `IEnumerable` ouvert sur le lecteur actif
`commandTimeout`	(facultatif) Le délai à utiliser sur la commande; `SqlMapper.Settings.CommandTimeout` n'est pas spécifié, `SqlMapper.Settings.CommandTimeout` est supposé (si spécifié)
`commandType`	Le type de commande en cours d'exécution; par défaut à `CommandText`

Remarques

La syntaxe pour exprimer les paramètres varie entre les SGBDR. Tous les exemples ci-dessus utilisent la syntaxe SQL Server, à savoir `@foo` ; Cependant `?foo` et `:foo` devraient également fonctionner correctement.

Examples

SQL paramétré de base

Dapper facilite le suivi des meilleures pratiques grâce à un SQL entièrement paramétré.

Les paramètres sont importants, donc dapper facilite la tâche. Vous venez d'exprimer vos paramètres de la manière habituelle pour votre SGBDR (généralement `@foo` `?foo` ou `:foo`) et donnez à dapper un objet qui *a un membre appelé* `foo` . La méthode la plus courante consiste à utiliser un type anonyme:

```
int id = 123;
string name = "abc";
connection.Execute("insert [KeyLookup](Id, Name) values(@id, @name)",
    new { id, name });
```

Et c'est tout. Dapper ajoutera les paramètres requis et tout devrait fonctionner.

Utiliser votre modèle d'objet

Vous pouvez également utiliser votre modèle d'objet existant en tant que paramètre:

```
KeyLookup lookup = ... // some existing instance
connection.Execute("insert [KeyLookup](Id, Name) values(@Id, @Name)", lookup);
```

Dapper utilise le texte de commande pour déterminer les membres de l'objet à ajouter - il n'ajoutera généralement pas de choses inutiles telles que `Description` , `IsActive` , `CreationDate` car la commande que nous avons publiée ne les implique pas - bien qu'il y ait des cas où pourrait le faire, par exemple si votre commande contient:

```
// TODO - removed for now; include the @Description in the insert
```

Il ne tente pas de comprendre que ce qui précède n'est qu'un commentaire.

Procédures stockées

Les paramètres des procédures stockées fonctionnent exactement de la même manière, sauf que Dapper ne peut pas tenter de déterminer ce qui doit / ne doit pas être inclus - tout ce qui est disponible est traité comme un paramètre. Pour cette raison, les types anonymes sont généralement préférés:

```
connection.Execute("KeyLookupInsert", new { id, name },
```

```
    commandType: CommandType.StoredProcedure);
```

Valeur Inlining

Parfois, la commodité d'un paramètre (en termes de maintenance et d'expressivité) peut être compensée par son coût en termes de performances pour le traiter en tant que paramètre. Par exemple, lorsque la taille de la page est fixée par un paramètre de configuration. Ou une valeur de statut correspond à une valeur `enum` . Considérer:

```
var orders = connection.Query<Order>(@"
select top (@count) * -- these brackets are an oddity of SQL Server
from Orders
where CustomerId = @customerId
and Status = @open", new { customerId, count = PageSize, open = OrderStatus.Open });
```

Le seul paramètre *réel* ici est `customerId` - les deux autres sont des pseudo-paramètres qui ne changeront pas réellement. Souvent, le SGBDR peut faire un meilleur travail s'il les détecte comme des constantes. Dapper a une syntaxe spéciale pour ceci - `{=name}` au lieu de `@name` - qui *ne* s'applique *qu'aux* types numériques. (Cela minimise toute surface d'attaque de l'injection SQL). Voici un exemple:

```
var orders = connection.Query<Order>(@"
select top {=count} *
from Orders
where CustomerId = @customerId
and Status = {=open}", new { customerId, count = PageSize, open = OrderStatus.Open });
```

Dapper remplace les valeurs par des littéraux avant d'émettre le code SQL. Le SGBDR voit donc quelque chose comme:

```
select top 10 *
from Orders
where CustomerId = @customerId
and Status = 3
```

Ceci est particulièrement utile lorsque vous autorisez les systèmes SGBDR non seulement à prendre de meilleures décisions, mais aussi à ouvrir des plans de requête que les paramètres réels empêchent. Par exemple, si un prédicat de colonne est associé à un paramètre, un index filtré avec des valeurs spécifiques sur ces colonnes ne peut pas être utilisé. Cela est dû au fait que la requête *suivante* peut avoir un paramètre différent de l'une de ces valeurs spécifiées.

Avec des valeurs littérales, l'optimiseur de requête peut utiliser les index filtrés car il sait que la valeur ne peut pas être modifiée dans les requêtes futures.

Extensions de liste

Un scénario courant dans les requêtes de base de données est `IN (...)` où la liste est générée au moment de l'exécution. La plupart des SGBDR ne possèdent pas une bonne métaphore pour cela - et il n'existe pas de solution universelle *de RDBMS* pour cela. Au lieu de cela, Dapper fournit une

extension automatique des commandes en douceur. Tout ce qui est nécessaire est une valeur de paramètre fournie qui est `IEnumerable` . Une commande impliquant `@foo` est étendue à `(@foo0,@foo1,@foo2,@foo3)` (pour une séquence de 4 éléments). L'utilisation la plus courante serait `IN` :

```
int[] orderIds = ...
var orders = connection.Query<Order>(@"
select *
from Orders
where Id in @orderIds", new { orderIds });
```

Cela se développe ensuite automatiquement pour émettre le code SQL approprié pour l'extraction à plusieurs lignes:

```
select *
from Orders
where Id in (@orderIds0, @orderIds1, @orderIds2, @orderIds3)
```

les paramètres `@orderIds0` etc. étant ajoutés en tant que valeurs extraites de l'array. Notez que le fait qu'il ne soit pas valide à l'origine est intentionnel, pour garantir que cette fonctionnalité n'est pas utilisée par erreur. Cette fonctionnalité fonctionne également correctement avec l'indicateur de requête `OPTIMIZE FOR` / `UNKNOWN` dans SQL Server; si tu utilises:

```
option (optimize for
    (@orderIds unknown))
```

il étendra ceci correctement à:

```
option (optimize for
    (@orderIds0 unknown, @orderIds1 unknown, @orderIds2 unknown, @orderIds3 unknown))
```

Effectuer des opérations contre plusieurs ensembles d'entrées

Parfois, vous voulez faire la même chose plusieurs fois. Dapper prend cela en charge sur la méthode `Execute` si le paramètre le *plus à* l' *extérieur* (qui est généralement un seul type anonyme ou une instance de modèle de domaine) est réellement fourni sous la forme d'une séquence `IEnumerable` . Par exemple:

```
Order[] orders = ...
// update the totals
connection.Execute("update Orders set Total=@Total where Id=@Id", orders);
```

Ici, dapper ne fait qu'une simple boucle sur nos données, essentiellement comme si nous l'avions fait:

```
Order[] orders = ...
// update the totals
foreach(Order order in orders) {
    connection.Execute("update Orders set Total=@Total where Id=@Id", order);
```

```
}
```

Cette utilisation devient *particulièrement* intéressante lorsqu'elle est associée à l'API `async` sur une connexion explicitement configurée pour tous les "multiples ensembles de résultats actifs". Dans cette utilisation, dapper va automatiquement *canaliser* les opérations, vous ne payez donc pas le coût de latence par ligne. Cela nécessite une utilisation légèrement plus compliquée,

```
await connection.ExecuteAsync(
    new CommandDefinition(
        "update Orders set Total=@Total where Id=@Id",
         orders, flags: CommandFlags.Pipelined))
```

Notez, cependant, que vous pourriez également vouloir examiner des paramètres de valeur de table.

Paramètres pseudo-positionnels (pour les fournisseurs qui ne prennent pas en charge les paramètres nommés)

Certains fournisseurs ADO.NET (notamment: OleDB) ne prennent pas en charge les paramètres *nommés* ; les paramètres sont plutôt spécifiés uniquement par la *position* , avec le `?` placeplace. Dapper ne saurait pas quel membre utiliser, Dapper permet donc une syntaxe alternative, `?foo?` ; ce serait la même chose que `@foo` ou `:foo` dans d'autres variantes SQL, sauf que dapper **remplacera** complètement le jeton de paramètre par `?` avant d'exécuter la requête.

Cela fonctionne en combinaison avec d'autres fonctionnalités telles que l'extension de la liste, de sorte que ce qui suit est valide:

```
string region = "North";
int[] users = ...
var docs = conn.Query<Document>(@"
     select * from Documents
     where Region = ?region?
     and OwnerId in ?users?", new { region, users }).AsList();
```

Les membres `.region` et `.users` sont utilisés en conséquence et le `.users` SQL émis est (par exemple, avec 3 utilisateurs):

```
     select * from Documents
     where Region = ?
     and OwnerId in (?,?,?)
```

Notez cependant que dapper **ne** permet **pas d'** utiliser le même paramètre plusieurs fois lors de l'utilisation de cette fonctionnalité. Cela permet d'éviter d'avoir à ajouter plusieurs fois la même valeur de paramètre (qui pourrait être importante). Si vous devez vous référer à la même valeur plusieurs fois, envisagez de déclarer une variable, par exemple:

```
declare @id int = ?id?; // now we can use @id multiple times in the SQL
```

Si des variables ne sont pas disponibles, vous pouvez utiliser des noms de membres en double

dans les paramètres - cela rendra également évident que la valeur est envoyée plusieurs fois:

```
int id = 42;
connection.Execute("... where ParentId = $id0$ ... SomethingElse = $id1$ ...",
      new { id0 = id, id1 = id });
```

Lire Paramètre Syntaxe Référence en ligne: https://riptutorial.com/fr/dapper/topic/10/parametre-syntaxe-reference

Chapitre 8: Paramètres dynamiques

Examples

Utilisation de base

Il n'est pas toujours possible de regrouper soigneusement tous les paramètres dans un seul objet / appel. Pour vous aider avec des scénarios plus complexes, dapper permet au paramètre `param` d'être une instance `IDynamicParameters` . Si vous faites cela, votre méthode `AddParameters` personnalisée est appelée au moment approprié et a transmis la commande à ajouter. Dans la plupart des cas, cependant, il suffit d'utiliser le type `DynamicParameters` préexistant:

```
var p = new DynamicParameters(new { a = 1, b = 2 });
p.Add("c", dbType: DbType.Int32, direction: ParameterDirection.Output);
connection.Execute(@"set @c = @a + @b", p);
int updatedValue = p.Get<int>("@c");
```

Ceci montre:

- (optionnel) population d'un objet existant
- (facultatif) ajouter des paramètres supplémentaires à la volée
- passer les paramètres à la commande
- récupérer toute valeur mise à jour une fois la commande terminée

Notez qu'en raison de la façon dont les protocoles de SGBDR fonctionnent, il est généralement fiable pour obtenir des valeurs de paramètres mis à jour **après** des données (à partir d' une `Query` opération ou QueryMultiple`) a été **entièrement** consommée (par exemple, sur SQL Server, les valeurs des paramètres mises à jour sont à la *fin* du flux TDS).

Paramètres dynamiques dans Dapper

```
connection.Execute(@"some Query with @a,@b,@c", new
{a=somevalueOfa,b=somevalueOfb,c=somevalueOfc});
```

Utiliser un objet modèle

Vous pouvez utiliser une instance d'objet pour former vos paramètres

```
public class SearchParameters {
  public string SearchString { get; set; }
  public int Page { get; set; }
}

var template= new SearchParameters {
  SearchString = "Dapper",
  Page = 1
};
```

```
var p = new DynamicParameters(template);
```

Vous pouvez également utiliser un objet anonyme ou un `Dictionary`

Lire Paramètres dynamiques en ligne: https://riptutorial.com/fr/dapper/topic/12/parametres-dynamiques

Chapitre 9: Requête de base

Syntaxe

- public static IEnumerable <T> Query <T> (cet IDbConnection cnn, chaîne sql, objet param = null, SqlTransaction transaction = null, bool buffered = true)
- public statique IEnumerable <dynamique> Query (cet IDbConnection cnn, chaîne sql, objet param = null, SqlTransaction transaction = null, bool buffered = true)

Paramètres

Paramètre	Détails
CNN	Votre connexion à la base de données, qui doit déjà être ouverte.
sql	Commande à exécuter.
param	Objet pour extraire les paramètres de.
transaction	Transaction dont cette requête fait partie, le cas échéant.
tamponné	S'il faut ou non mettre en mémoire tampon les résultats de la requête. Ceci est un paramètre facultatif avec la valeur par défaut étant true. Lorsque la mise en mémoire tampon est vraie, les résultats sont mis en mémoire tampon dans une `List<T>` , puis renvoyés sous la forme d'un `IEnumerable<T>` sûr pour une énumération multiple. Lorsque la mise en mémoire tampon est fausse, la connexion SQL est maintenue ouverte jusqu'à ce que vous ayez fini de lire, ce qui vous permet de traiter une seule ligne à la fois en mémoire. Plusieurs énumérations engendreront des connexions supplémentaires à la base de données. Bien que false mis en mémoire tampon soit très efficace pour réduire l'utilisation de la mémoire si vous ne gérez que de très petits fragments d'enregistrements renvoyés, il se caractérise par une surcharge de performances considérable par rapport à la matérialisation rapide du jeu de résultats. Enfin, si vous avez de nombreuses connexions SQL non tamponnées simultanées, vous devez tenir compte de la famine du pool de connexions, ce qui entraîne le blocage des requêtes jusqu'à ce que les connexions soient disponibles.

Examples

Interrogation pour un type statique

Pour les types connus à la compilation, utilisez un paramètre générique avec `Query<T>` .

```
public class Dog
{
    public int? Age { get; set; }
    public Guid Id { get; set; }
    public string Name { get; set; }
    public float? Weight { get; set; }

    public int IgnoredProperty { get { return 1; } }
}

//
IDBConnection db = /* ... */;

var @params = new { age = 3 };
var sql = "SELECT * FROM dbo.Dogs WHERE Age = @age";

IEnumerable<Dog> dogs = db.Query<Dog>(sql, @params);
```

Interrogation pour les types dynamiques

Vous pouvez également interroger dynamiquement si vous omettez le type générique.

```
IDBConnection db = /* ... */;
IEnumerable<dynamic> result = db.Query("SELECT 1 as A, 2 as B");

var first = result.First();
int a = (int)first.A; // 1
int b = (int)first.B; // 2
```

Requête avec paramètres dynamiques

```
var color = "Black";
var age = 4;

var query = "Select * from Cats where Color = :Color and Age > :Age";
var dynamicParameters = new DynamicParameters();
dynamicParameters.Add("Color", color);
dynamicParameters.Add("Age", age);

using (var connection = new SqlConnection(/* Your Connection String Here */))
{
    IEnumerable<dynamic> results = connection.Query(query, dynamicParameters);
}
```

Lire Requête de base en ligne: https://riptutorial.com/fr/dapper/topic/3/requete-de-base

Chapitre 10: Résultats multiples

Syntaxe

- public static SqlMapper.GridReader QueryMultiple (ce cnn IDbConnection, chaîne SQL, objet param = null, transaction IDbTransaction = null, int? commandTimeout = null, CommandType? commandType = null)
- public static SqlMapper.GridReader QueryMultiple (cette commande IDbConnection cnn, CommandDefinition)

Paramètres

Paramètre	Détails
CNN	Votre connexion à la base de données doit déjà être ouverte
sql	La chaîne sql à traiter contient plusieurs requêtes
param	Objet pour extraire les paramètres de
SqlMapper.GridReader	Fournit des interfaces pour lire plusieurs ensembles de résultats à partir d'une requête Dapper

Examples

Exemple de base de résultats multiples

Pour extraire plusieurs grilles en une seule requête, la méthode `QueryMultiple` est utilisée. Cela vous permet ensuite de récupérer chaque grille de *manière séquentielle* via des appels successifs sur le `GridReader` renvoyé.

```
var sql = @"select * from Customers where CustomerId = @id
            select * from Orders where CustomerId = @id
            select * from Returns where CustomerId = @id";

using (var multi = connection.QueryMultiple(sql, new {id=selectedId}))
{
   var customer = multi.Read<Customer>().Single();
   var orders = multi.Read<Order>().ToList();
   var returns = multi.Read<Return>().ToList();
}
```

Lire Résultats multiples en ligne: https://riptutorial.com/fr/dapper/topic/8/resultats-multiples

Chapitre 11: Tables Temp

Examples

Table temporaire qui existe tant que la connexion reste ouverte

Lorsque la table temporaire est créée par elle-même, elle restera pendant que la connexion est ouverte.

```
// Widget has WidgetId, Name, and Quantity properties
public async Task PurchaseWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        await conn.OpenAsync();

        await conn.ExecuteAsync("CREATE TABLE #tmpWidget(WidgetId int, Quantity int)");

        // populate the temp table
        using(var bulkCopy = new SqlBulkCopy(conn)) {
            bulkCopy.BulkCopyTimeout = SqlTimeoutSeconds;
            bulkCopy.BatchSize = 500;
            bulkCopy.DestinationTableName = "#tmpWidget";
            bulkCopy.EnableStreaming = true;

            using(var dataReader = widgets.ToDataReader())
            {
                await bulkCopy.WriteToServerAsync(dataReader);
            }
        }

        await conn.ExecuteAsync(@"
            update w
            set Quantity = w.Quantity - tw.Quantity
            from Widgets w
                join #tmpWidget tw on w.WidgetId = tw.WidgetId");
    }
}
```

Comment travailler avec des tables temporaires

Le point concernant les tables temporaires est qu'elles sont limitées à la portée de la connexion. Dapper ouvrira et fermera automatiquement une connexion si elle n'est pas déjà ouverte. Cela signifie que toute table temporaire sera perdue directement après sa création, si la connexion passée à Dapper n'a pas été ouverte.

Cela ne fonctionnera pas:

```
private async Task<IEnumerable<int>> SelectWidgetsError()
{
  using (var conn = new SqlConnection(connectionString))
  {
    await conn.ExecuteAsync(@"CREATE TABLE #tmpWidget(widgetId int);");
```

```
    // this will throw an error because the #tmpWidget table no longer exists
    await conn.ExecuteAsync(@"insert into #tmpWidget(WidgetId) VALUES (1);");

    return await conn.QueryAsync<int>(@"SELECT * FROM #tmpWidget;");
  }
}
```

En revanche, ces deux versions fonctionneront:

```
private async Task<IEnumerable<int>> SelectWidgets()
{
  using (var conn = new SqlConnection(connectionString))
  {
    // Here, everything is done in one statement, therefore the temp table
    // always stays within the scope of the connection
    return await conn.QueryAsync<int>(
      @"CREATE TABLE #tmpWidget(widgetId int);
        insert into #tmpWidget(WidgetId) VALUES (1);
        SELECT * FROM #tmpWidget;");
  }
}

private async Task<IEnumerable<int>> SelectWidgetsII()
{
  using (var conn = new SqlConnection(connectionString))
  {
    // Here, everything is done in separate statements. To not loose the
    // connection scope, we have to explicitly open it
    await conn.OpenAsync();

    await conn.ExecuteAsync(@"CREATE TABLE #tmpWidget(widgetId int);");
    await conn.ExecuteAsync(@"insert into #tmpWidget(WidgetId) VALUES (1);");
    return await conn.QueryAsync<int>(@"SELECT * FROM #tmpWidget;");
  }
}
```

Lire Tables Temp en ligne: https://riptutorial.com/fr/dapper/topic/6594/tables-temp

Chapitre 12: Transactions

Syntaxe

- conn.Execute (sql, transaction: tran); // spécifie le paramètre par nom
- conn.Execute (sql, paramètres, tran);
- conn.Query (sql, transaction: tran);
- conn.Query (sql, paramètres, tran);
- attend conn.ExecuteAsync (sql, transaction: tran); // Async
- attendez conn.ExecuteAsync (sql, paramètres, tran);
- attendez conn.QueryAsync (sql, transaction: tran);
- attendez conn.QueryAsync (sql, paramètres, tran);

Examples

Utiliser une transaction

Cet exemple utilise SqlConnection, mais toute IDbConnection est prise en charge.

De plus, toute IDbTransaction est prise en charge à partir de la IDbConnection associée.

```
public void UpdateWidgetQuantity(int widgetId, int quantity)
{
    using(var conn = new SqlConnection("{connection string}")) {
        conn.Open();

        // create the transaction
        // You could use `var` instead of `SqlTransaction`
        using(SqlTransaction tran = conn.BeginTransaction()) {
            try
            {
                var sql = "update Widget set Quantity = @quantity where WidgetId = @id";
                var parameters = new { id = widgetId, quantity };

                // pass the transaction along to the Query, Execute, or the related Async
methods.
                conn.Execute(sql, parameters, tran);

                // if it was successful, commit the transaction
                tran.Commit();
            }
            catch(Exception ex)
            {
                // roll the transaction back
                tran.Rollback();

                // handle the error however you need to.
                throw;
            }
        }
    }
}
```

Accélérer les inserts

Enrouler un groupe d'insertions dans une transaction les accélérera en fonction de cette question / réponse StackOverflow .

Vous pouvez utiliser cette technique ou vous pouvez utiliser Bulk Copy pour accélérer une série d'opérations connexes à effectuer.

```
// Widget has WidgetId, Name, and Quantity properties
public void InsertWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        conn.Open();

        using(var tran = conn.BeginTransaction()) {
            try
            {
                var sql = "insert Widget (WidgetId,Name,Quantity) Values(@WidgetId, @Name,
@Quantity)";
                conn.Execute(sql, widgets, tran);
                tran.Commit();
            }
            catch(Exception ex)
            {
                tran.Rollback();
                // handle the error however you need to.
                throw;
            }
        }
    }
}
```

Lire Transactions en ligne: https://riptutorial.com/fr/dapper/topic/6601/transactions

Chapitre 13: Utiliser Async

Examples

Appeler une procédure stockée

```
public async Task<Product> GetProductAsync(string productId)
{
    using (_db)
    {
        return await _db.QueryFirstOrDefaultAsync<Product>("usp_GetProduct", new { id =
productId },
            commandType: CommandType.StoredProcedure);
    }
}
```

Appeler une procédure stockée et ignorer le résultat

```
public async Task SetProductInactiveAsync(int productId)
{
    using (IDbConnection con = new SqlConnection("myConnectionString"))
    {
        await con.ExecuteAsync("SetProductInactive", new { id = productId },
            commandType: CommandType.StoredProcedure);
    }
}
```

Lire Utiliser Async en ligne: https://riptutorial.com/fr/dapper/topic/1353/utiliser-async

Chapitre 14: Utiliser DbGeography et DbGeometry

Examples

Configuration requise

1. installez l'assembly `Microsoft.SqlServer.Types` requis; ils ne sont pas installés par défaut et sont disponibles auprès de Microsoft ici en tant que "Types de CLR Microsoft® System pour Microsoft® SQL Server® 2012" - notez qu'il existe des programmes d'installation distincts pour x86 et x64.

2. installez `Dapper.EntityFramework` (ou son équivalent nommé fort); Cela peut se faire via l'interface utilisateur "Manage NuGet Packages ..." de l'EDI ou (dans la console du gestionnaire de packages):

```
install-package Dapper.EntityFramework
```

3. ajouter les redirections de liaison d'assembly requises; En effet, Microsoft est livré avec les assemblys v11, mais Entity Framework demande la version 10; Vous pouvez ajouter ce qui suit à `app.config` ou `web.config` sous l'élément `<configuration>` :

```
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <dependentAssembly>
      <assemblyIdentity name="Microsoft.SqlServer.Types"
          publicKeyToken="89845dcd8080cc91" />
      <bindingRedirect oldVersion="10.0.0.0" newVersion="11.0.0.0" />
    </dependentAssembly>
  </assemblyBinding>
</runtime>
```

4. Dites "dapper" à propos des nouveaux gestionnaires de types disponibles, en ajoutant (quelque part dans votre démarrage, avant d'essayer d'utiliser la base de données):

```
Dapper.EntityFramework.Handlers.Register();
```

Utiliser la géométrie et la géographie

Une fois les gestionnaires de types enregistrés, tout devrait fonctionner automatiquement et vous devriez pouvoir utiliser ces types comme paramètres ou valeurs de retour:

```
string redmond = "POINT (122.1215 47.6740)";
DbGeography point = DbGeography.PointFromText(redmond,
    DbGeography.DefaultCoordinateSystemId);
DbGeography orig = point.Buffer(20); // create a circle around a point
```

```
var fromDb = connection.QuerySingle<DbGeography>(
  "declare @geos table(geo geography); insert @geos(geo) values(@val); select * from @geos",
  new { val = orig });

Console.WriteLine($"Original area: {orig.Area}");
Console.WriteLine($"From DB area: {fromDb.Area}");
```

Lire Utiliser DbGeography et DbGeometry en ligne:
https://riptutorial.com/fr/dapper/topic/3984/utiliser-dbgeography-et-dbgeometry

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Dapper.NET	Adam Lear, balpha, Community, Eliza, Greg Bray, Jarrod Dixon, Kevin Montrose, Matt McCabe, Nick, Rob, Shog9
2	Exécution des commandes	Adam Lear, Jarrod Dixon, Sklivvz, takrl
3	Gestionnaires de types	Benjamin Hodgson, Community, Marc Gravell
4	Inserts en vrac	jhamm
5	Manipulation des Nulls	Marc Gravell
6	Multimapping	Devon Burriss
7	Paramètre Syntaxe Référence	4444, Marc Gravell, Nick Craver
8	Paramètres dynamiques	Marc Gravell, Matt McCabe, Meer
9	Requête de base	Adam Lear, Chris Marisic, Cigano Morrison Mendez, Community, cubrr, Jarrod Dixon, jrummell, Kevin Montrose, Matt McCabe
10	Résultats multiples	Marc Gravell, Yaakov Ellis
11	Tables Temp	jhamm, Rob, takrl
12	Transactions	jhamm
13	Utiliser Async	Dean Ward, Matt McCabe, Nick, Woodchipper
14	Utiliser DbGeography et DbGeometry	Marc Gravell