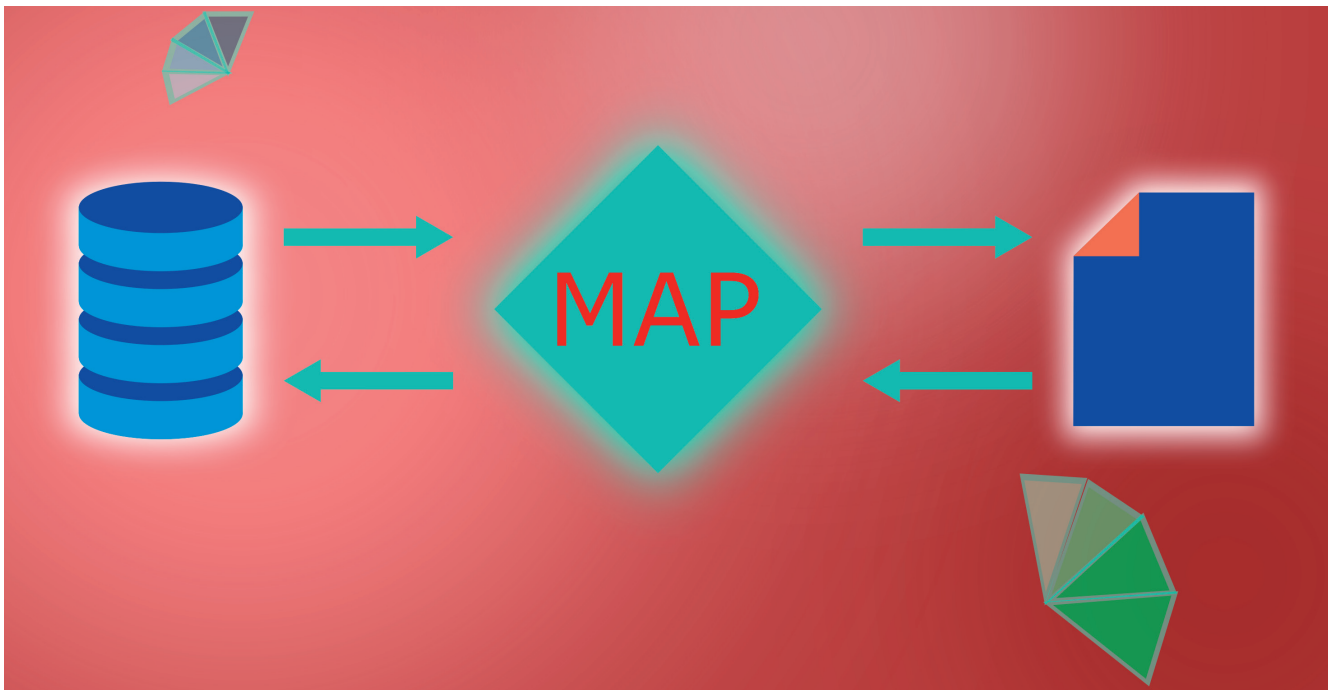


Bild: dotnetpro / OrMapper

DATENBANKZUGRIFFE MIT DAPPER

# Der König der Micro-ORMs?

Datenbankzugriffe und Objekt-Mapping sind möglich, doch mit der Funktionsvielfalt des Entity Framework kann Dapper nicht mithalten. Trotzdem lohnt ein Blick darauf.

Daten aus einer relationalen Datenbank in die Anwendung und zurück zu transportieren: Diese Anforderung ist so alt wie Datenbanken selbst. Das relationale Modell einer Datenbank passt nicht gut auf das objektorientierte der Anwendung, was immer wieder zu Kopfzerbrechen führt. Stichwort Impedance Mismatch (object-relational impedance mismatch). Mittlerweile wurde das Problem vielschichtig angegangen und mal mehr, mal weniger gut gelöst. Falls eine relationale Datenbank nicht zwingend erforderlich ist, kann eine andere zum Einsatz kommen, bei der das Problem weniger hervortritt.

Muss es aber eine relationale Datenbank sein, kann einer der zahlreichen Objekt-Relationalen-Mapper (ORMs) helfen, die das Problem automatisiert lösen. Strukturen der Datenbank werden als Klassenstrukturen generiert und SQL-Anweisungen automatisch zur Laufzeit erzeugt, wenn mit der Datenbank interagiert werden soll.

Auch wenn die ersten Versionen von ORMs alles andere als perfekt waren und die Performance oftmals noch Bauchschmerzen hervorruft, funktioniert das in vielen Fällen vollautomatisch und beachtlich gut.

Allerdings sind ORMs wahre Komplexitätsmonster. Nicht zwingend in der Verwendung, aber die vielen Automatismen und die Notwendigkeit, mit vielen Datenbanken zusammenarbeiten zu können, fordern ihren Tribut im Sinne von sehr großen Frameworks.

Wer einmal ORMs beziehungsweise das Mapping von Daten einer Datenquelle in die Programmiersprache in Perfektion sehen möchte, der werfe einen Blick auf F# Type Provider. Dort funktioniert der Zugriff auf Daten mit einer nicht gekannten Leichtigkeit.

Dieser Artikel widmet sich der Bibliothek Dapper, einem kleineren ORM, der eine geringere Komplexität besitzt als die größeren Verwandten.

## Das ist Dapper

Über die Jahre sind viele ORM-Frameworks entstanden, die sich hauptsächlich im Funktionsumfang unterscheiden. Mit den Daten aus SQL-Anweisungen Objekte füllen (Mapping) können sie praktisch alle. Unterschiede ergeben sich aber in Funktionen wie dem Zwischenspeichern von Ergebnissen, dem Erfassen von Änderungen an Objekten, der Erzeugung von SQL-Anweisungen und dem Lazy Loading. Ausgewachsene ORMs bieten das in der Regel an. Die kleineren Vertreter, gerne Micro-ORMs genannt, sparen sich einen Großteil dieser Komplexität.

Womit wir bei der Bibliothek Dapper sind. Auf die Frage, ob Dapper ein ORM ist, wird gerne mit „Jein“ geantwortet. Oft

wird das Projekt als „König der C#-Micro-ORMs“ beschrieben. Andere bezeichnen es eher als Objekt-Mapper für .NET.

Diese Kategorisierung kommt daher, dass Dapper immer dann gerne eingesetzt wird, wenn die Daten von SQL-Anweisungen zwar auf .NET-Objekte gemappt werden sollen, ansonsten die Kontrolle aber beim Entwickler liegen soll – zum Beispiel, weil die für Abfragen notwendigen SQL-Anweisungen manuell geschrieben werden. Damit sind sehr feine Performance-Optimierungen möglich, falls das notwendig ist.

Das bedeutet aber auch, dass Dapper die typischen Features eines ORM nicht mitbringt. SQL-Queries werden zu Objekten gemappt – das war's. Es gibt kein Caching, kein Change Tracking, keine SQL-Erzeugung, keine Datenbankmigrationen und kein Identity Management. Für das Change Tracking und die SQL-Erzeugung gibt es zwar Erweiterungen, die rüsten aber nur eine minimale Funktionalität nach – verglichen mit ORMs wie zum Beispiel Entity Framework. Wer damit zufrieden ist oder diesen eingeschränkten Funktionsumfang schätzt, der kann mit Dapper sehr weit kommen.

Dapper kommt aus dem Hause Stack Exchange (Stack Overflow). In der NuGet-Galerie sind zwar die Autoren Marc Gravell und Nick Craver gelistet, die Bibliothek wird aber vom gesamten Team gepflegt. Die Bibliothek scheint recht beliebt zu sein. Die Downloads über alle Versionen hinweg übersteigen die 30 Millionen, mit bereits über einer Million Downloads für die Version 2.0.30, die beim Schreiben dieses Artikels die aktuelle ist.

## Installation

Die Installation von Dapper ist recht ereignislos. Unterstützt werden das .NET Framework und .NET Core. In der NuGet-Galerie werden 844 Pakete gelistet, wenn nach Dapper gesucht wird. Darunter sind zahlreiche Erweiterungen. Dapper ist Open Source, der Code steht auf GitHub [1] zur Verfügung und steht unter der Apache License Version 2.0.

## Warum Dapper einsetzen?

Die primäre Frage ist, warum Dapper in einem Projekt zum Einsatz kommen sollte. Das Angebot an umfangreicheren ORMs ist nicht gerade klein, und falls sich die Anforderungen doch einmal ändern sollten, kann das größere ORM mithalten, wo Dapper eventuell nicht mehr hinterherkommt.

Wer Dapper einsetzen möchte, sollte sich die Hauptfunktion in Erinnerung rufen: Performance. Die Bibliothek ist entstanden, um die maximale Geschwindigkeit bei Abfragen herauszuholen. Für die Entwickler von Dapper, die Betreiber des Netzwerks Stack Exchange mit Plattformen wie Stack Overflow, war das wichtig, denn sie nutzten seinerzeit LINQ to SQL, was nicht die notwendige Performance mitbrachte, um dem steigenden Traffic Herr zu werden. Daher schrieben sie ihren eigenen Micro-ORM.

SQL-Anweisungen werden bei Dapper nicht erst erzeugt oder durch eine Abstraktionsschicht vorgegeben, sondern direkt im Code hinterlegt. Es können zwar parametrisierte Abfragen genutzt werden, was bezüglich SQL Injection wichtig ist, aber ansonsten ist es ein ziemlich direkter Zugriff auf die Datenbank. Das ist bereits ein handfester Grund, Dapper nicht einzusetzen. Denn wer keine SQL-Anweisungen schreiben mag oder den Prozess aus anderen Gründen abstrahieren möchte, wird mit Dapper nicht glücklich.

Ganz allgemein gilt Dapper als eine gute Wahl für Szenarien, in denen sich Read-only-Daten häufig ändern und diese ebenso häufig gelesen werden müssen.

Auch in zustandslosen Anwendungen macht Dapper eine sehr gute Figur. Des Weiteren macht Dapper keine Annahmen über die Datenbankstruktur, da die SQL-Anweisungen manuell geschrieben werden müssen. Das ist in allen Szenarien wichtig, wo die Struktur austauschbar ist, sich häufig ändert oder zum Beispiel nicht nach gültigen Standards und Konventionen normalisiert wurde.

Der Einsatz von Dapper kann auch dann schwierig werden, wenn ein vorhandenes Projekt Richtung Dapper migriert werden soll. Wenn dieses Projekt bereits einen ORM wie zum Beispiel das Entity Framework einsetzt und stark von den angebotenen Features Gebrauch macht, kann der Umstieg auf Dapper zumindest eine Geduldsprobe werden. Dapper nutzt, wie bereits angesprochen, ausschließlich Plain Old Clr Objects (POCOs) – also kein Change Tracking. Dieses Feature für sich betrachtet zurückzubauen, beziehungsweise durch andere Mechanismen zu ersetzen, kann ein Kraftakt sein.

## Unterstützte Datenbankprovider

Dapper unterstützt zahlreiche Datenbanken. Die Arbeit mit den Datenbankprovidern wurde in Erweiterungen ausgelagert, sodass neue hinzukommen können, wenn sich die Landschaft der Datenbanken ändert. Eine vollständige Liste der unterstützten Datenbanken muss der Autor an dieser Stelle schuldig bleiben, da keine aufzutreiben war. Wer eine kennt, darf sich mit sachdienlichen Hinweisen gerne melden. Eine Suche in der NuGet-Galerie hat ergeben, dass die bekannten Datenbanken wie zum Beispiel MSSQL, MySQL, Oracle und PostgreSQL unterstützt werden. Darüber hinaus bietet Dapper Erweiterungen für die *IDbConnection*-Schnittstelle an, was weitere Möglichkeiten bietet. Die Aufgabe, mit dem jeweiligen Datenbankprovider kompatible SQL-Anweisungen zu schreiben, fällt eh in das Gebiet des Entwicklers.

## Lesen von Daten

Die Bibliothek bietet einen Satz an Methoden, um Ergebnisse der Datenbank abzufragen. Diese sind unter anderem:

- *Query*
- *QueryFirst*
- *QueryFirstOrDefault*
- *QuerySingle*
- *QuerySingleOrDefault*
- *QueryMultiple*

Das folgende Snippet zeigt den Zugriff auf eine Tabelle:

```
using (var connection = new SqlConnection(...))
{
  var authors = connection.Query<Author>(
    "Select * From Author").ToList();
}
```

Der Code ist recht simpel gehalten und zeigt den Aufbau der Datenbankverbindung und die *Query*-Methode von Dapper. Intern wird das Ergebnis auf eine Klasse gemappt, die in diesem Beispiel wie folgt aussehen könnte:

```
public class Author
{
  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
}
```

Bei der Implementierung müssen wir uns nicht um das Mapping der Ergebnismenge auf die POCO-Klasse kümmern. Auch die Umwandlung zu einer Liste wird uns abgenommen. Die SQL-Anweisung wurde allerdings manuell von uns hinterlegt. Diese paar Zeilen Code verdeutlichen die Kernaufgabe von Dapper ziemlich gut. Die anderen genannten Methoden sind Abwandlungen der *Query*-Methode, zum Beispiel, um nur den ersten Eintrag der Ergebnisse zu erhalten oder einen Default-Wert, falls gar nichts zurückgeliefert wird. Die Vorgehensweise bleibt aber die gleiche.

Bei den SQL-Anweisungen werden Parameter unterstützt. Sie können entweder von anonymer oder dynamischer Natur sein, beziehungsweise als Liste oder Zeichenkette angegeben werden. Im anonymen Fall sieht das beispielsweise wie folgt aus:

```
using (var connection = new SqlConnection(...))
{
  var author = connection.Query<Author>(
    "Select * From Author WHERE Id = @Id",
    new { Id = id }).SingleOrDefault();
}
```

Bei einer Parameterliste könnten SQL-Anweisungen mit einem *IN*-Befehl mit Abfragedaten gefüllt werden. Nehmen wir als Beispiel die folgende SQL-Anweisung:

```
var sql = "SELECT * FROM Invoice WHERE Kind IN @Kind;";
```

Hierbei kann die Ausführung folgendermaßen aussehen:

```
connection.Query<Invoice>(sql, new {Kind = new[] {
  InvoiceKind.StoreInvoice, InvoiceKind.WebInvoice}})
  .ToList();
```

Der Bereich des Schlüsselworts *IN* wird korrekt von Dapper mit Daten befüllt, sodass die Abfrage parametrisiert ist und ohne großen Aufwand ausgeführt werden kann.

## INSERT, UPDATE und DELETE

Die vorherigen Beispiele haben die Abfrage von Daten mit den Query-Methoden angesprochen. Zusätzlich existieren bei Dapper Methoden, um SQL-Anweisungen auszuführen, die Änderungen an Daten verursachen. Mit der *Execute*-Methode können Stored Procedures, *INSERT*-, *UPDATE*- und *DELETE*-Anweisungen ausgeführt werden. Die Parametrisierung von Daten funktioniert auch in diesem Fall. Im folgenden Beispiel wird eine *UPDATE*-Anweisung ausgeführt, um einen Autoren-Datensatz in der entsprechenden Tabelle zu aktualisieren. Die Daten kommen direkt aus einem Autor-Objekt und werden von Dapper gemappt.

```
using (var connection = new SqlConnection(...))
{
  var query = "UPDATE Author SET FirstName = @FirstName,
    LastName = @LastName WHERE Id = @Id";

  var rowsAffected = connection.Execute(query, author);
}
```

Gleichzeitig wird die Anzahl der betroffenen Zeilen zurückgegeben.

## Mapping von Daten

Das Mapping von Daten ist bereits in den vorherigen Beispielen kurz angesprochen worden. Zusätzlich zu den gezeigten Mapping-Varianten gibt es noch einige mehr. Daten können zu anonymen Objekten gemappt werden oder zu konkreten Klassen, wie bereits gezeigt. Des Weiteren unterstützt Dap-

**Listing 1: Multi-Mapping der Ergebnisse einer SQL-Anweisung mit INNER JOIN**

```
string sql = "SELECT * FROM Invoice AS A INNER JOIN InvoiceDetail AS B ON A.InvoiceID = B.InvoiceID;";
using (var connection = new SqlConnection(...))
{
  var invoices = connection.Query<Invoice, InvoiceDetail, Invoice>(
    sql,
    (invoice, invoiceDetail) =>
    {
      invoice.InvoiceDetail = invoiceDetail;
      return invoice;
    },
    splitOn: "InvoiceID")
    .Distinct()
    .ToList();
}
```

per das Multi-Mapping. Dabei können mit einer SQL-Anweisung und einem eingebauten *INNER JOIN* die Daten der zweiten Tabelle in Objekte überführt werden. Listing 1 zeigt dazu ein etwas umfangreicheres Beispiel. Dieses Feature von Dapper kann die Arbeit mit verschachtelten SQL-Anweisungen und verschachtelten Rückgabedaten stark vereinfachen.

## Performance

Um die Geschwindigkeit zu messen, gibt es ein eigenes Benchmark-Projekt [2], das viele der Funktionen gegen andere ORMs überprüft. Getestet werden zum Beispiel LINQ to DB, manuell geschriebene SQL-Anweisungen, Belgrade, PetaPoco, Entity Framework und Entity Framework Core sowie NHibernate. Die konkreten Daten sind in einer umfangreichen Liste [3] aufgeführt. Dapper schneidet dabei grundsätzlich sehr gut ab.

## Erweiterungen

Dass es zu Dapper zahlreiche Erweiterungen gibt, wurde weiter oben im Artikel kurz angerissen. Zum Beispiel sind die unterschiedlichen Datenbankprovider als Erweiterungen realisiert. Darüber hinaus gibt es noch einige weitere, darunter sind sowohl kostenfreie als auch kostenpflichtige Erweiterungen. Die meisten zielen darauf ab, Dapper Funktionen zu verpassen, die in ausgewachsenen ORMs zu finden sind. Wie sinnvoll das ist, sich einem mehr oder weniger Standard-Feature-Set eines ORM von dieser Seite zu nähern und nicht direkt ein ausgewachsenes ORM zu nutzen, muss jeder für sich entscheiden. Ein Vorteil dieser Variante ist, dass nur die Funktionen nachgerüstet werden, die tatsächlich im Projekt benötigt werden. Insgesamt bleibt Dapper dann sehr wahrscheinlich immer noch kleiner als andere ORMs.

Beispiele für diese Erweiterungen sind Dapper.Async, Dapper.FluentMap, Dapper.Rainbow, Dapper.SimpleCRUD und andere.

Dapper.Async rüstet das Interface *IDbConnection* mit asynchronen Methoden nach, die im ursprünglichen Dapper-Funktionsumfang fehlen. Nach der Installation der Erweiterung gibt es Methoden wie *ExecuteAsync*, *QueryAsync*, *QuerySingleAsync* und dergleichen.

Die Erweiterung Dapper.FluentMap fügt Funktionen hinzu, die ähnlich zu anderen Mapping-Bibliotheken sind. Sie erinnern stellenweise an Funktionen von AutoMapper und Co. Nach der Installation können Eigenschaften zum Beispiel gemappt oder ignoriert werden:

```
Map(i => i.InvoiceID).ToColumn("Id");
Map(i => i.Detail).Ignore();
```

Mit der Erweiterung Dapper.Rainbow hält eine abstrakte Klasse Einzug ins Projekt. Sie dient als Basisklasse für alle eigenen Klassen, die mit Dapper interagieren sollen. Dadurch werden CRUD-Operationen wie INSERT, UPDATE und DELETE als Methoden zur Verfügung gestellt. Nach der Installation der Erweiterung muss eine Container-Klasse erstellt werden, die von der neu hinzugekommenen abstrakten Klasse erbt. Dort sind alle Tabellen als Eigenschaft zu hinterlegen:

```
public Table<Invoice> Invoices { get; set; }
```

Von da an sind Zugriffe in der Art *db.Invoices.Insert()* und *db.Invoices.Get()* möglich. Leider besitzt diese Erweiterung keine gute Dokumentation und einige Limitierungen. Zusammengesetzte Schlüssel sind nicht möglich und die Spalte mit der Identität muss *Id* heißen.

## Dapper Plus

Neben den genannten Erweiterungen gibt es noch Dapper Plus. Es gibt zwar eine kostenfreie Testversion, im Grunde handelt es sich aber um eine kostenpflichtige Erweiterung. Hinzugefügt werden Funktionen zu Bulk-Operationen bei *INSERT*, *UPDATE*, *DELETE* und *MERGE*, um große Datenmengen mit diesen Operationen zu verarbeiten. Diese Funktionen sind in Dapper ansonsten nicht vorhanden.

## Fazit

Dapper [4] ist eine spannende Bibliothek. Die Funktionen sind im Vergleich mit größeren ORMs definitiv eingeschränkt. Dapper behauptet aber auch gar nicht, ein ausgewachsener ORM zu sein. Das Projekt kann mit dem Titel „König der C#-Micro-ORMs“ zufrieden sein, den die Community ihm verliehen hat. Optimiert auf Performance, macht Dapper seine ihm zugewiesene Aufgabe gut. Wer mehr möchte, muss sich durch die angebotenen Erweiterungen wühlen oder doch ein größeres ORM nutzen.

Darüber hinaus ist die Dokumentation ordentlich und der Code wird regelmäßig gepflegt. Zusätzlich gibt es zahlreiche Tutorials [5], die ebenfalls gut erklären, wie Dapper eingesetzt werden kann. Das rundet das Gesamtpaket ab.

Wer ein schnelles Micro-ORM sucht, das die Mapping-Aufgabe abnimmt, aber die Kontrolle der SQL-Anweisungen ermöglicht, ist mit Dapper gut beraten. Insgesamt ist Dapper einen Blick wert und hat sich ein „Sehr gut“ und eine Empfehlung verdient. ■

*[1] Dapper als Git Repository auf GitHub, www.dotnetpro.de/SL2003Frameworks1*
*[2] Performance-Projekt für ORM im Dapper-Repository, www.dotnetpro.de/SL2003Frameworks2*
*[3] Übersicht über Performance-Messungen verschiedener ORMs, www.dotnetpro.de/SL2003Frameworks3*
*[4] Website von Dapper, www.dotnetpro.de/SL2003Frameworks4*
*[5] Tutorial zu Dapper, https://dapper-tutorial.net/*

**Fabian Deitelhoff**
promoviert am Graduiertenkolleg „User-Centred Social Media“ im Themenumfeld der Code Comprehension. Zudem ist er Autor, Entwickler, Trainer und Gründer von www.brickobotik.de. Erreichbar über www.fabiandeitelhoff.de oder **@FDeitelhoff**

**dnpCode** A2003Frameworks