RELATIONALES MAPPING MIT DATAOBJECTS.NET

Die Datenbrücke

DataObjects.NET soll performant sein und einen Business Logic Layer (BLL) bieten.

Relationale Daten auf der einen, Klassen und Typsystem auf der anderen Seite. Und dazwischen ein guter objektrelationaler Mapper (ORM). So stellt sich in vielen Projekten die Datenhaltung dar.

Von ORMs gibt es einige. Dapper wurde beispielsweise in der dotnetpro-Ausgabe 3/2020 [1] als Vertreter eines sehr performanten Micro-ORM vorgestellt. NHibernate [2] ist eine sehr bekannte Lösung, und Entity Framework (EF) erfreut sich auch unter .NET Core einer immer größeren Beliebtheit.

Die Liste von ORMs ließe sich noch deutlich erweitern, da über die Jahre und Jahrzehnte viele Lösungen entstanden sind.

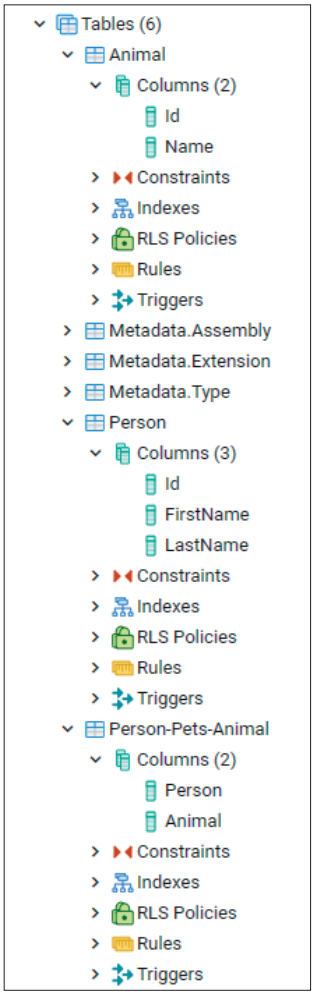
Warum also taucht hier in der Kolumne Frameworks und mehr eine weitere Lösung in die Richtung OR-Mapping auf?

Zum einen einfach aus Interesse, wie eine andere Bibliothek das Problem des objektrelationalen Mappings angeht.

Zum anderen, weil DataObjects.NET [3], um das es in diesem Artikel geht, viele teils sehr vollmundige Versprechungen auf der Website macht.

Ein weiterer Grund ist, dass es bereits seit 2003 existiert und vor Kurzem erst eine neue Version erschienen ist.

Über diesen Zeitraum haben sich sicherlich viele Best Practices herausgebildet, und es scheint eine aktive Nutzerbasis zu geben.



Übersicht über das automatisch generierte Datenbank-Schema (Tabellen) mit den jeweiligen Spalten (Bild 1)

Was ist DataObjects.NET?

Neben DataObjects.NET wird ein Business Logic Layer (BLL) angeboten. Daher findet die Kategorisierung auch als Framework und nicht als Bibliothek statt, denn eine Schicht für die Business-Logik hat einen enormen Einfluss darauf, wie das eigene Projekt gestaltet werden muss.

Auf der Website wird das Projekt als „Development Framework" für moderne .NET-Projekte angepriesen, das insbesondere mit nichttrivialen Domain-Modellen umgehen können soll, den Code-First-Ansatz auf die Spitze treibt und eine skalierbare und performante Lösung bereitstellt.

DataObjects.NET kommt aus einer Zeit, als das Rapid Application Development (RAD) ganz hoch im Kurs stand. Daher legt das Framework viel Wert darauf, dass möglichst wenig direkt konfiguriert werden muss und wenig Code zu schreiben ist.

Von Closed Source zu Open Source

Eine interessante Entwicklung hat DataObjects.NET bereits ganz offensichtlich durchgemacht. Seit der Version 6, die im Januar 2020 erschienen ist, wird das Framework als Open Source unter der MIT-Lizenz angeboten. Für die vorherigen Versionen 4.x und 5.x, die weiterhin zur Verfügung stehen und zumindest bei der Version 5 auch weiterhin gepflegt werden, sieht das anders aus. DataObjects.NET ist nämlich als Closed-Source-Projekt mit einem kostenpflichtigen Subscription-Modell gestartet.

Die Preise [4] finden sich auf der Website. Diese gelten weiterhin für die Versionen 4 und 5. Wenn diese in Zukunft genutzt werden sollen, ist wie bisher eine jährliche Lizenz zu erwerben. Das ist eine interessante Entwicklung, weil es wahrscheinlich Druck ausübt, die Version 6.0 zu nutzen. Die Release-Übersicht [5] auf GitHub listet auch ältere Versionen, zum Beispiel für den 5.x-Branch.

Über die Motive für den Wechsel zu Open Source kann an dieser Stelle nur spekuliert werden. Eine Recherche dazu hat nichts ergeben und eine Anfrage ist bis zur Fertigstellung dieses Artikels unbeantwortet geblieben. Der Blog [6] zum Framework ist eingeschlafen.

Angemerkt sei auch, dass relationale Datenbanksysteme nur eine der möglichen Lösungen sind, um Daten abzulegen. Um das Problem des objektrelationalen Mappings zu lösen beziehungsweise zu umgehen, haben sich diverse Techniken und Systeme herausgebildet.

Installation

Die Installation ist wie immer simpel. Die kostenpflichtigen Versionen von DataObjects.NET wurden und werden von Xtensive LLC [7] vertrieben, einem Unternehmen rund um Cloud-Lösungen, AI, Data Science, .NET, Web, Mobile und IoT. Das hat zur Folge, dass alle Pakete für DataObjects.NET mit *Xtensive* beginnen und auch so in der NuGet-Galerie und Konsorten gelistet sind. Das Hauptpaket ist *Xtensive.Orm* [8]. Alle anderen Pakete sind Erweiterungen beziehungsweise

unterstützte Datenbank-Systeme. Bei der Installation ist eventuell der Hinweis wichtig, für welche Version ein Paket existiert. Es scheint so, als seien manche Pakete bei der Version 5.1.0-Beta stehen geblieben, zum Beispiel *Xtensive.Orm.SqlServerCe*. Ob das so bleibt und die Erweiterungen aufgegeben werden oder ob noch Änderungen kommen, bleibt eine offene Frage. Einer der hauptverantwortlichen Entwickler ist Alex Kulakov [9], der laut GitHub-Profil bei Xtensive LLC angestellt ist und sich mindestens um das Mergen von Pull Requests kümmert.

Aktuell ist Version 6.0.4 vom 22. Dezember 2020. Unterstützt werden das .NET Framework und .NET Core. Wie bereits beschrieben ist das Projekt ab Version 6 Open Source. Der Code steht unter der MIT-Lizenz und ist auf GitHub [10] verfügbar.

Ein Blick auf die Unternehmenswebsite zeigt im Übrigen, dass sie zuletzt im Jahr 2018 erweitert wurde. Das deckt sich mit den letzten Blogposts, die vom Oktober 2018 stammen. Insgesamt gibt das ein etwas merkwürdiges Bild ab, weil zeitgleich das Framework DataObjects.NET Open Source und kostenfrei ist und es weiterhin Updates gibt.

Die Kernfunktionen

DataObjects.NET tritt mit einer ganzen Reihe von Features an. Die folgende Liste beschreibt einige der Kernfunktionen des Frameworks:

Listing 1: Klasse Person für DataObjects.NET

```
namespace Twainsoft.Articles
    .dnp.DataObjectsNET.Model
{

  [HierarchyRoot]
  public class Person : Entity
  {

    public Person(string firstName, string lastName)
    {
      FirstName = firstName;
      LastName = lastName;
    }

    [Key] [Field] public int Id { get; set; }

    [Field] public string FirstName { get; set; }

    [Field] public string LastName { get; set; }

    [Field]
    public EntitySet<Animal> Pets
     { get; private set; }
  }
}
```

Wechsel von Visual Studio zu Rider

Ein kleiner Einschub zur Kolumne Frameworks und mehr: Ab jetzt werden die Beispiele, wenn es Sinn ergibt, ein Beispielprojekt anzubieten, mit JetBrains Rider erstellt und nicht mehr mit Visual Studio. Die Projektdateien sind aber vollständig kompatibel. Der Wechsel ist eher eine Geschmacksentscheidung gewesen.

- Umfangreiche Integration in verschiedene Entwicklungsumgebungen wie Visual Studio 2019+, Visual Studio Code und Rider (über die Unterstützung von MSBuild-Targets).
- Ansatz des Domain-Modelling durch den Code-First-Ansatz (Attribut-basiertes Mapping).
- Zahlreiche unterstützte .NET-Datentypen, persistente Auflistungen, deklarative Business-Regeln und zahlreiche Lifecycle-Events.
- Automatische Pflege von Datenbankschemata.
- Unterstützung von Queries, LINQ, hohe Performance und viele Datenbanksysteme wie MS SQL Server, MySQL, Oracle, PostgreSQL und Firebird.

Laut Eigenwerbung des Frameworks werden insbesondere der Code-First-Ansatz für das Domain-Modelling, die Business-Regeln und die hohe Performance hervorgehoben. Das mag alles seine Richtigkeit haben, in Zeiten eines immer besser werdenden Entity Frameworks und mit NHibernate im Rücken ist der Vorsprung in den letzten Jahren aber eher geschrumpft als größer geworden. Was nicht bedeutet, dass die zahlreichen Funktionen des Frameworks unnötig sind. Auf den ersten Blick macht DataObjects.NET nämlich einen sehr guten Eindruck.

Eine umfangreiche Liste der primären Features des Frameworks ist auf der Website des Projekts [11] zu finden.

Das Testprojekt

Zur Begleitung des Artikels wurde ein minimales Testprojekt erstellt. Basis ist .NET Core 3.1. Mit der zunächst anvisierten Version 13 von PostgreSQL gab es allerdings Schwierigkeiten. Anscheinend ist ein Update noch nicht bei DataObjects.NET angekommen.

Beim Konfigurieren der Domäne mit den Modellen wird mit PostgreSQL 13 reproduzierbar die folgende Exception erzeugt:

```
System.AggregateException: One or more errors occurred.
(42703: Spalte a.consrc existiert nicht)  --->
Npgsql.PostgresException (0x80004005): 42703: Spalte
a.consrc existiert nicht
```

Ein Hinweis [12] auf Stack Overflow brachte dann die notwendige Inspiration, es mit der Version 11 von PostgreSQL zu versuchen. Damit funktionierte das Testprojekt mit dem entsprechenden Datenbankprovider [13] anschließend ein- ▶

wandfrei. Die Konfiguration der Datenbank beziehungsweise die Prüfung der Daten fand mit pgAdmin 4 statt.

Ein Modell erstellen

Als erster Schritt bei der Arbeit mit DataObjects.NET steht die Konfiguration des Modells an. Diese Modelle werden dann zu einer Domäne registriert, um die Daten zum Beispiel persistieren zu können. Es ist auch möglich, zuerst die Domäne und den Datenspeicher zu konfigurieren, was viele andere Tutorials genauso machen. Da die Modelle dort aber registriert werden müssen, wie der Artikel im nachfolgenden Abschnitt zeigt, ist es sinnvoll, ebendiese Modelle zuerst zu definieren.

Das Framework DataObjects.NET bietet eine spannende Arbeitsweise, wie sich bei der Verknüpfung von Modellen mit der Domäne im nächsten Abschnitt zeigt. Modelle werden als normale Klassen definiert und über Attribute ausgezeichnet, sodass DataObjects.NET weiß, wie es die jeweiligen Eigenschaften handhaben soll.

Listing 1 zeigt das am Beispiel einer einfachen Entität namens *Person*. Das Feld *ID* ist als Schlüssel ausgewiesen, alle anderen als einfache Datenfelder. Mit der kleinen Ausnahme der Haustiere, weil der Typ *EntitySet<Animal>* angibt, dass es eine Collection ist. Die Modell-Klasse *Animal* ist ähnlich aufgebaut, besitzt aber nur eine ID und einen Namen.

Das ist das generelle Vorgehen bei DataObjects.NET. Persistente Daten werden über Klassen modelliert, die von der Oberklasse *Entity* erben müssen. Persistente Daten werden mit dem *[Field]*-Attribut ausgezeichnet, und die erste Klasse in einer Hierarchie, beziehungsweise die Root-Klasse, muss mit *[HierarchyRoot]* ausgezeichnet werden.

Darüber hinaus gibt es weitere Möglichkeiten, Datenklassen auszuzeichnen und das letztendliche Persistieren der Daten zu beeinflussen. Es kann zum Beispiel mit dem Attribut *[Index(<Name>)]* ein Index zu einem Feld definiert werden. Über die Basisklasse ist die Definition einer Struktur möglich:

```
public class Point : Structure
{
  [Field]
  public int X { get; set; }

  [Field]
  public int Y { get; set; }
}
```

Dieser Typ *Point* lässt sich daraufhin als Feld zu einer Datenklasse hinzufügen. Eine Struktur hat keine eigene Identität, was bedeutet, dass keine eigene Tabelle angelegt wird. Die beiden Eigenschaften *X* und *Y* der Point-Klasse werden in die Datenklasse als Spalten repliziert. Das ist ein Beispiel dafür, wie die Wiederverwendbarkeit im objektorientierten Code erhöht werden kann.

Domänen und Datenspeicher

Als nächster Schritt steht die Konfiguration der Domäne und des Speicherorts für die Daten an. DataObjects.NET muss wissen, mit welchen Modellen es umgehen soll. Diese Modelle sind zwar als Klassen definiert, müssen aber noch zu einer Domäne zusammengefasst und auf diese Weise registriert werden.

Listing 2: Konfiguration von Domäne und Modellen

```
var config = new DomainConfiguration("postgresql://
    postgres:root@localhost:5433/dnp-article")
{
  UpgradeMode = DomainUpgradeMode.Recreate
};

config.Types.Register(typeof(Person).Assembly);
config.Types.Register(typeof(Animal).Assembly);
var domain = Domain.Build(config);
```

Listing 2 zeigt, wie das vonstattengeht. Die *DomainConfiguration* nimmt einen Connection-String entgegen. Wie eingangs erwähnt, basiert das Beispiel auf PostgreSQL, sodass die Zeichenkette für den Verbindungsaufbau darauf ausgelegt ist. Die Angabe des Ports ist optional und rührt daher, dass das der Server für die PostgreSQL-Version 11 ist, da es mit der aktuellen Version 13 nicht funktioniert, wie ebenfalls eingangs erwähnt wurde. Der allgemeine Aufbau des Connection-Strings lautet wie folgt:

```
postgresql://<user>:<password>@<server>
  :<port>/<database>
```

Der *UpgradeMode* ist in diesem Beispiel auf *Recreate* eingestellt. Das bedeutet, dass bei jeder Ausführung des Codes dieser Domäne die Datenbank vollständig neu aufgebaut wird. Alle Tabellen, Daten und sonstigen Informationen gehen verloren. Das ist für den Projektstart beziehungsweise für eine neue Domäne mit neuen Modellen ein guter Modus. Für spätere Versionen einer Anwendung und Datenbank dann allerdings nicht mehr. Neben der *Recreate*-Einstellung existieren noch die folgenden Optionen:

- *Validate*
- *Skip*
- *Perform*
- *PerformSafely*
- *LegacyValidate* und *LegacySkip*

Die Skip-Option ist die Haudrauf-Methode. Es findet keine Prüfung der Kompatibilität oder ein Upgrade des Schemas statt. Es wird einfach angenommen, dass beide Schema-Versionen kompatibel sind. Dem Feature des Schema-Upgrades ist in der Dokumentation ein umfangreicher Abschnitt [14] gewidmet, der die technischen Details klärt.

Listing 2 zeigt weiterhin, wie die Modell-Klassen bei der Domänen-Konfiguration registriert werden müssen. Es wird der Typ der Klasse genutzt und über die Register-Methode hinzugefügt. Ein abschließender Aufruf von *Domain.Build()* er-

zeugt die Domäne mit den registrierten Modellen, die anschließend einsatzbereit ist.

Was jetzt für alle deutlich werden sollte, die schon einmal mit objektrelationalen Mappern gearbeitet haben: Wo ist der Code zum Mapping? Oder anders gefragt: Braucht es noch weiteren Code zum Mapping von Daten und Objekten? Die Antwort ist nein, das ist nicht erforderlich. DataObjects.NET erzeugt diesen Glue-Code, wie er zuweilen genannt wird, durch die Attribute an den Klassen ganz automatisch.

Das Framework nutzt dafür Aspect Oriented Programming (AOP), um die notwendigen Operationen für die Persistenz- und die Business-Logik-Schicht in die annotierten Klassen zu injizieren. Diese Aspekte liegen in Form von Methoden oder Delegaten vor, die am Anfang oder Ende von anderen Methoden automatisch aufgerufen werden.

Das Injizieren erfolgt nach dem Kompilieren der Assembly im IL-Code (Intermediate Language). DataObjects.NET nutzt das Framework PostSharp [15], um den Mapping-Code zu injizieren. Das macht DataObjects.NET auf der einen Seite sehr elegant, weil viel Code automatisch erzeugt wird und das eigene Projekt schlanker bleibt. Auf der anderen Seite ist in Nicht-Standardsituationen eventuell mehr Konfigurationsaufwand notwendig, damit die Automatismen beim Erzeugen der Aspekte den eigenen Wünschen entsprechend angepasst werden können. Die Nutzung von APO wird noch bei der Anwendung der Datenbank-Operationen auffallen, wie der nächste Abschnitt zeigt.

Datenbank-Operationen

Wenn die Domäne und die Modelle erstellt und Letztere registriert sind, sind beide Teilbereich einsatzbereit. Jetzt ist es an der Zeit, Datenbank-Operationen auszuführen. Zum Beispiel Daten erzeugen und in der Datenbank ablegen. Listing 3 zeigt das anhand eines kleinen Beispiels. Zur Domäne muss zunächst eine Session geöffnet werden. Diese dann zu aktivieren sollte laut Dokumentation nicht notwendig sein. Ohne den Aufruf schlägt das Beispiel aber fehl mit der Meldung, dass keine aktive Session vorhanden ist. Anschließend wird eine Transaktion geöffnet. Nun folgt das Erzeugen der Daten, indem die Modell-Klassen genutzt werden, um Objekte zu erstellen. Ein Aufruf der *Complete*-Methode schließt diese Transaktion ab. Ein Blick in die Datenbank zeigt, dass die Tabellen erzeugt wurden – die *Person*- und *Animal*-Tabellen ebenso wie die Verknüpfungstabelle und Tabellen für Metadaten (siehe Bild 1).

Listing 3: Operationen mittels aktiver Session

```
using (var session = domain.OpenSession())
{

  using (session.Activate())
  {
    using (var transactionScope =
      session.OpenTransaction())
    {
      var p1 = new Person("Test 1", "Person 1");

      p1.Pets.Add(new Animal("Bella"));
      p1.Pets.Add(new Animal("Lotte"));

      var p2 = new Person("Test 2", "Person 2");
      p2.Pets.Add(new Animal("Mischa"));

      transactionScope.Complete();
    }
  }
}
```

Der Beispielcode überrascht: Es gibt keine offensichtliche Verknüpfung zwischen den Datenklassen des Modells und der Domäne. Es werden zwar brav aus der Domäne eine Session und daraus eine Transaktion gestartet, aber die Datenklassen für die beiden Personen und die drei Tiere stehen visuell etwas losgelöst im Code herum.

Auch an dieser Stelle greift AOP. Die Verknüpfung zwischen Transaktion und Modellklassen erfolgt automatisch im Hintergrund beziehungsweise im IL-Code. Das ist auf der einen Seite gewöhnungsbedürftig, auf der anderen Seite verschlankt es den Code und erlaubt auf elegante Art und Weise die Implementierung von Datenbank-Operationen.

Die Abfrage von Daten

Wo Daten gespeichert werden, ist die Anforderung nicht weit, diese abzufragen und zum Beispiel zu löschen. Listing 4 zeigt die Abfrage von Personen mit einem bestimmten Nachnamen über LINQ. Bei den Beispieldaten ist das nur eine Person, die anschließend aus der Session und nach Abschluss der umschließenden Transaktion auch aus der Datenbank gelöscht wird, inklusive der Zuordnung eines Haustiers, ohne das Tier aber selbst zu löschen, da das nicht konfiguriert wurde. Die Datenabfrage und der Löschvorgang funktionieren beide über die aktive Session.

Mehr ist nicht notwendig, um auf die Daten zuzugreifen. DataObjects.NET bietet die Standardfunktionen von .NET an, wie beispielsweise Standard-C#-Operatoren, Methoden für den Vergleich, String-Methoden und dergleichen. Projektionen sind ebenso möglich wie lokale Collections in Queries, Joins, Vererbung und Sub-Queries, sowie vorkompilierte Queries, um eine eventuell längere Laufzeit zu vermeiden. Damit stehen viele Möglichkeiten bereit, auf die Daten zuzugreifen, die mit DataObjects.NET gespeichert wurden.

Business-Regeln mittels Validierung

Wie eingangs erwähnt, kommt DataObjects.NET mit dem Feature des Business Logic Layers (BLL). Zum BLL gehört noch die Validierungs-Schicht: Jede Entität unterliegt einer fortlaufenden Validierung. Das kann in kritischen Regionen des Codes beziehungsweise bei kritischen Modell-Entitäten durchaus relevant sein. Oft ist es aber ausreichend, Entitäten zu validieren, wenn die Transaktion committet wird. Jede Transaktion besitzt ihren eigenen *ValidationContext*. Mo- ▶

dell-Klassen, die von Entity erben, implementieren das *IValidationAware*-Interface, um eine Validierung zu ermöglichen. Falls explizit keine Validierung durchgeführt werden soll, kann diese deaktiviert werden:

```
using (var inconsistencyRegion =
    session.DisableValidation()) {

  // Mit der Entität arbeiten...

  inconsistencyRegion.Complete();
}
```

Wenn dauerhaft eine Validierung on-demand erfolgen soll, kann das der Domänen-Konfiguration mitgeteilt werden:

```
domainConfiguration.ValidationMode =
  ValidationMode.OnDemand;
```

Bei der Validierung wird zwischen einem Object-Level und einem Property-Level unterschieden. Bei Ersterem kommt das eben angesprochene *IValidationAware*-Interface zum Tragen. Eine Modell-Klasse könnte zum Beispiel folgende Validierungsregel beinhalten:

```
protected override void OnValidate()
{
  base.OnValidate();

  if (IsSubscribedToNewsletter &&
    string.IsNullOrEmpty(Email)) {
      throw new Exception("...");
  }
}
```

Dadurch lassen sich zwar die Eigenschaften unabhängig voneinander verändern, beispielsweise die E-Mail-Adresse löschen, je nach Konfiguration der Validierung wird aber spätestens beim Abschließen der Transaktion eine Exception geworfen.

Bei einer Validierung auf Basis der Felder einer Entität lassen sich Regeln per Attribut definieren, die dann bei einer Validierung automatisch überprüft werden können:

```
[LengthConstraint(Min = 2, Max = 128)]
[NotNullOrEmptyConstraint]
public string FirstName { get; set;}
```

Um die Arbeit mit diesen Validierungen zu erleichtern, bietet DataObjects.NET einige vordefinierte Einschränkungen an. Zum Beispiel ein *EmailConstraint* für E-Mail-Adressen oder ein *FutureConstraint*, um zu prüfen, ob ein Datum in der Zukunft liegt. Wenn eigene Constraints notwendig sind, ist von der Klasse *PropertyConstraintAspect* zu erben, die alles Notwendige bereitstellt. Generell bietet das in DataObjects.NET integrierte Validierungs-Framework viele Möglichkeiten, die Konsistenz von Daten zu prüfen.

Erweiterungen

Das Framework DataObjects.NET lässt sich zum Beispiel über das Paket für Bulk-Operationen erweitern, mit dem Modifikationen auf vollständige *IQueryable*-Instanzen anwendbar sind:

```
Query.All<Bar>()
  .Where(a => a.Id == 1)
  .Set(a => a.Count, 2)
  .Update();
```

Die Änderungen werden dann in entsprechende Server-seitige *UPDATE*- oder *DELETE*-SQL-Anweisungen übersetzt.

Darüber hinaus existieren Erweiterungen für die Lokalisierung beziehungsweise Internationalisierung, für Sicherheitsaspekte und um das Change-Tracking/Auditing zu ermöglichen. Die Pakete sind alle zum Beispiel in der NuGet-Galerie verfügbar und lassen sich recht simpel zum Projekt hinzufügen.

Etwas zur Performance

Bei DataObjects.NET gibt es einige Stellschrauben, um die beste Performance für den eigenen Anwendungsfall herauszukitzeln.

Statement Batching ist so eine Möglichkeit, um zum Beispiel Veränderungen an den Daten nicht als einzelne Statements, sondern in einem Schwung an den Server zu senden.

Lazy Loading ist eine weitere Stellschraube. Das Standardverhalten ist, dass zum Beispiel EntitySets und Structures automatisch per Lazy Loading geladen werden.

Des Weiteren erlaubt es das Prefetch API, mehrere Anfragen zu Entity oder EntitySet zu bündeln und in einer Query mehrere auszuführen. Das spart ebenfalls einige Roundtrips zum Datenbankserver.

Diese Möglichkeiten sind sehr ausführlich und mit detaillierten Codebeispielen in einem eigenen Abschnitt in der Dokumentation [16] beschrieben.

Listing 4: Abfrage von Daten mittels LINQ

```
using (var transactionScope =
  session.OpenTransaction())
{
  var person =
    from p in session.Query.All<Person>()
    where p.LastName == "Person 1"
    select p;

  Console.WriteLine(person.FirstOrDefault());

  session.Remove(person);

  transactionScope.Complete();
}
```

DataObjects.NET, Entity Framework und das Ökosystem

Aber ist DataObjects.NET im Vergleich zu anderen ORMs noch immer konkurrenzfähig? Die schlichte Antwort lautet ja. Die Kombination aus der guten Performance, der aspektorientierten Programmierung und dem eingebauten Validierungs-Framework macht Spaß und ergibt Sinn. Es spart tatsächlich einiges an Code.

Die deutlich differenziertere Antwort ist allerdings höchstens ein Jein. Die Möglichkeit, eine bereits bestehende Datenbank in Modelle zu überführen, scheint zu fehlen. Eine Recherche hat dazu zumindest nichts ergeben. Wenn das doch funktioniert, freuen wir uns über eine kurze Info.

Aktuell bedeutet das aber, dass eine Datenbank mit den Möglichkeiten von DataObjects.NET im Code nachmodelliert werden muss, was sehr viel Aufwand bedeuten kann.

Das Ökosystem scheint das Framework dagegen immer noch zu akzeptieren beziehungsweise auf dem Plan zu haben. Für ComponentOne gibt es beispielsweise für Windows Forms Erweiterungen für DataObjects.NET, sodass sich beide Welten sehr gut kombinieren lassen, um Oberflächen beziehungsweise ganze Anwendungen mit beiden Technologien zu erstellen.

Ob das für andere Technologien in der Zukunft auch so sein wird, zum Beispiel für die Windows Presentation Foundation (WPF), ist allerdings mehr als fraglich.

Aus alten Artikeln im Internet ist zu entnehmen, dass DataObjects.NET ursprünglich entstanden ist, da NHibernate und das Entity Framework nicht performant genug waren und nur eine partielle LINQ-Implementierung enthielten. Das hat sich in den vergangenen Jahren deutlich geändert. Insbesondere bei Entity Framework Core wird viel Aufwand investiert, um die Performance zu steigern. Darauf hat DataObjects.NET in den letzten Jahren viel an Vorsprung verloren und wird ihn auch weiterhin verlieren.

Ob es eine gute Idee war, das Projekt als Open Source zu veröffentlichen, wird sich daher zeigen müssen. Es besteht die Chance, dass die Community einspringt, die sich in den letzten Jahren um das Framework entwickelt hat. Eventuell ein anderes Unternehmen, das stark auf das Projekt setzt.

Fazit

Das Framework DataObjects.NET macht noch immer einen sehr guten Eindruck. Die Funktionalität stimmt, die Inbetriebnahme war kein großes Problem, abseits der Schwierigkeiten, die aber eher mit PostgreSQL zu tun hatten.

Die Magie, die durch die aspektorientierte Programmierung ins Spiel kommt, macht sich das Framework gut zu nutzen. Das wird bei der Implementierung eigener Modelle und Domänen schnell deutlich. Ob sich diese Automagie in größeren Projekten irgendwann rächt, steht allerdings auf einem anderen Blatt. Häufig kann so etwas passieren, wenn Verhaltensweisen nicht explizit modelliert werden.

Alles in allem ist DataObjects.NET ein sehr ordentliches und rundes Framework. Es ist dem Projekt deutlich anzumerken, dass schon einige Jahre an Arbeit hineingeflossen sind, die sich bis Version 5 auch haben monetarisieren lassen. Die Dokumentation ist sehr gut und bietet zahlreiche und sehr detaillierte Codebeispiele.

Allerdings war beim Schreiben dieses Artikels nur die Dokumentation zur Version 5 aufzufinden. Ob es eine spezielle Dokumentation zur aktuellen Version 6 gibt, hat auch eine Recherche nicht zutage gefördert.

Abschließend bleibt nur zu sagen, dass das Framework DataObjects.NET definitiv einen Blick wert ist. Es verdient sich die Note „Sehr gut“ und eine Empfehlung.

Meine persönliche Hoffnung ist ja, dass das Projekt nicht einfach in der Versenkung verschwindet, weil es jetzt nicht mehr monetarisiert wird, sondern dass sich im Zweifel die bereits aktive Community um das Framework kümmert und es angemessen pflegt und weiterentwickelt. ■

[1] Fabian Deitelhoff, Der König der Micro-ORMs?, dotnetpro 3/2020, Seite 98, www.dotnetpro.de/A2003Frameworks

[2] Die Website zu NHibernate, https://nhibernate.info

[3] Die Website zu DataObjects.NET, https://dataobjects.net

[4] Die Preise der Versionen 4.x und 5.x von DataObjects.NET, https://dataobjects.net/prices.aspx

[5] Die Release-Übersicht zu DataObjects.NET auf GitHub, www.dotnetpro.de/SL2103Frameworks1

[6] Der Blog zu DataObjects.NET, http://blog.dataobjects.net

[7] Die Website von Xtensive LLC, dem Unternehmen hinter DataObjects.NET, http://x-tensive.com

[8] Das Core-Paket zu DataObjects.NET in der NuGet-Galerie, www.dotnetpro.de/SL2103Frameworks2

[9] Das GitHub-Profil von Alex Kulakov, www.dotnetpro.de/SL2103Frameworks3

[10] Das Repository von DataObjects.NET auf GitHub, www.dotnetpro.de/SL2103Frameworks4

[11] Die Feature-Übersicht von DataObjects.NET, www.dotnetpro.de/SL2103Frameworks5

[12] Hinweis auf Stack Overflow zum PostgreSQL-Fehler, www.dotnetpro.de/SL2103Frameworks6

[13] Der Datenbankprovider für PostgreSQL, www.dotnetpro.de/SL2103Frameworks7

[14] Dokumentation zum Schema-Upgrade von DataObjects.NET, www.dotnetpro.de/SL2103Frameworks8

[15] Die Website zu PostSharp, www.postsharp.net

[16] Dokumentation zu Performance-Themen von DataObjects.NET, www.dotnetpro.de/SL2103Frameworks9

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 A2103Frameworks