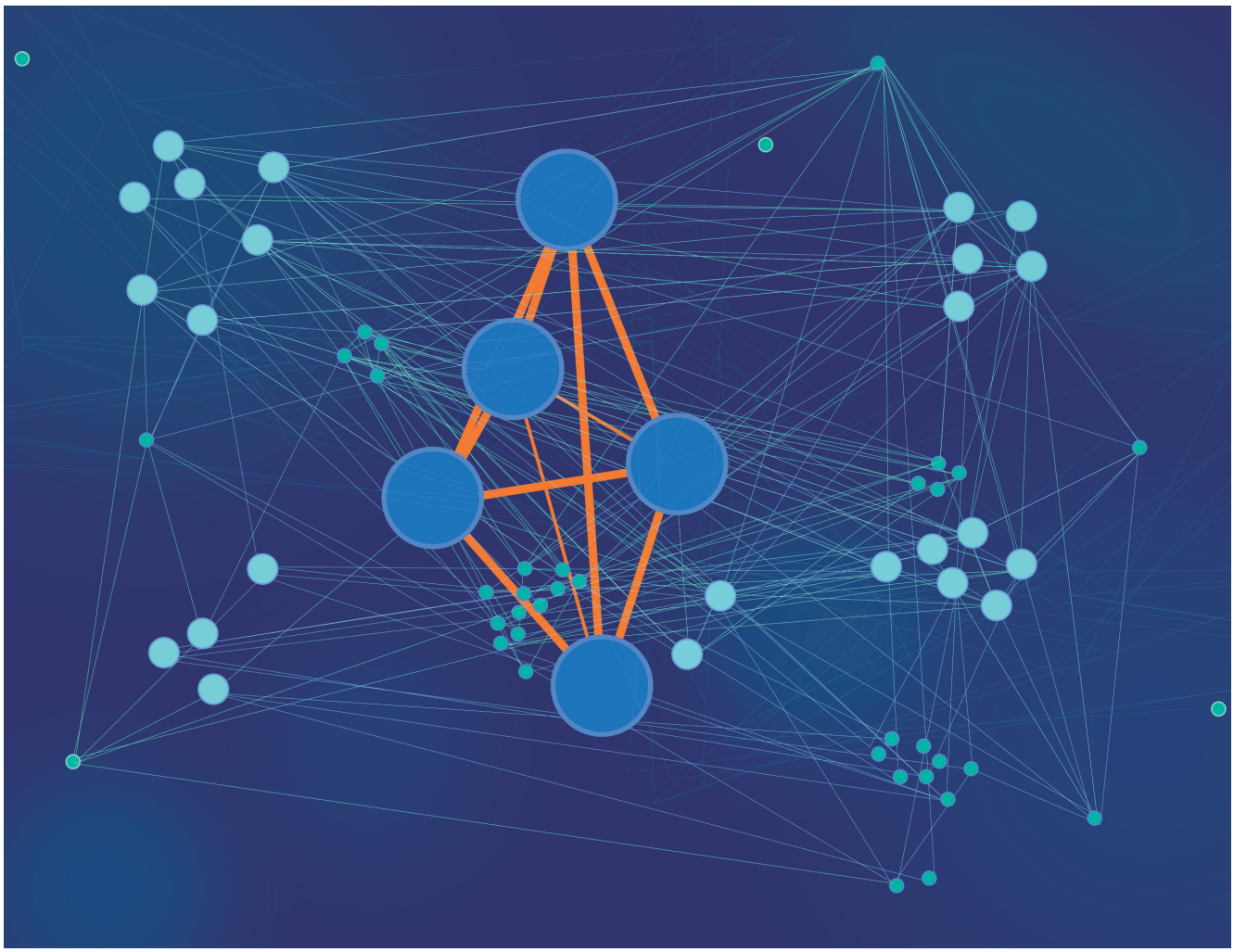


Bild: dotnetpro

GRAPHDATENBANKEN ABFRAGEN MIT EXRAM.GREMLINQ

Gremlin + LINQ = Gremlinq

Ein Object-Graph-Mapper überträgt das bewährte ORM-Konzept auf Graphdatenbanken.

Der Begriff „NoSQL" ist längst nicht mehr das Buzzword, das es vor gut einem Jahrzehnt einmal war: Zu umfassend etabliert haben sich mittlerweile die verschiedenen Ansätze, Datenmodelle jenseits des Relationalen und damit auch neue Abfragesprachen zu beschreiben. Die einfachste Ausprägung eines NoSQL-Konzepts ist sicherlich der Key-Value Store, der eine ansonsten unstrukturierte Sammlung von Schlüssel-Wert-Paaren darstellt. Etwas komplexer wird es bei einem Document Store, der sich darauf versteht, im Prinzip beliebig strukturierte Daten (Dokumente) zu indizieren und zu speichern. Den beiden genannten Konzepten gemeinsam ist, dass die kleinste Informationseinheit, die sie betrachten, zwar eine lokale Struktur aufweist, eine globale Struktur aber nicht gegeben ist – eine Menge an gespeicherten Schlüssel-Wert-Paaren oder Dokumenten steht ansonsten nicht zwingend in einer Beziehung zueinander. Graphbasierte NoSQL-Ansätze schließen nun diese Lücke, indem sie die gespeicherten Datenstrukturen aufeinander verweisen lassen, sie also miteinander in Relation setzen. Auf konzeptueller Ebene ergeben sich demnach Knoten und Kanten. Schlussendlich gibt es noch eine ganze Reihe weiterer Konzepte, auf die hier nicht eingegangen werden soll.

Relativ neu ist nun die wachsende Beliebtheit, derer sich gerade im „Database as a Service"-Bereich der graphbasierte Ansatz erfreut – so hat Microsoft mit Azure Cosmos DB erst seit Ende 2017 eine gehostete Graphdatenbank im Angebot [1], ebenso verhält es sich mit IBMs Variante des Graphdatenbanksystems JanusGraph [2]. Amazon Web Services ist mit Neptune seit 2018 am Start [3].

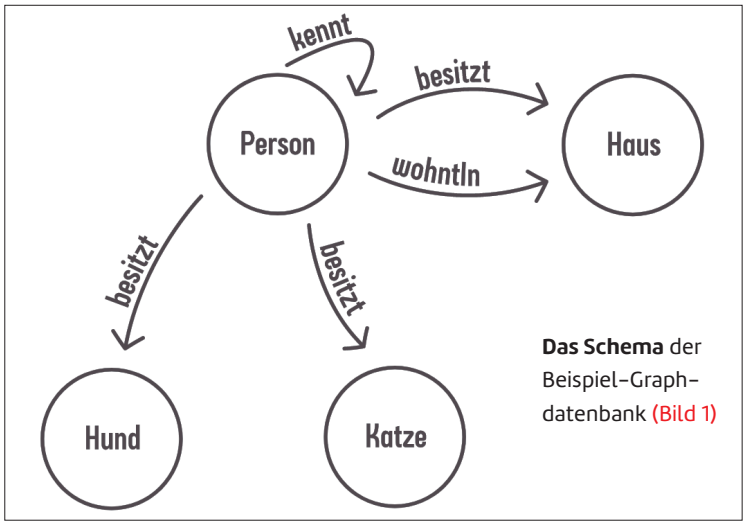
Was jedoch die .NET-Backend-Welt betrifft, so scheinen Graphdatenbanken immer noch in einer Nische festzustecken – Zeit, dass sich das ändert! ▶

Der vorliegende Artikel wird zuerst auf die Vor- und Nachteile des graphbasierten Ansatzes eingehen und die Abfragesprache Gremlin vorstellen. Anschließend werden Sie schnell und unkompliziert eine lokale Graphdatenbank zum Laufen bringen, auf die Sie dann von einer C#-Console-App zugreifen. Dazu wird der Object-Graph-Mapper ExRam.Gremlinq vorgestellt, mit dessen Hilfe sich der Zugriff auf die Datenbank einfach und typsicher bewerkstelligen lässt.

Graph? Wieso? Weshalb? Warum?

Warum nun eine Graphdatenbank? Einmal abgesehen von der Herausforderung, etwas Neues abseits der erprobten relationalen Datenspeicher auszuprobieren, lohnt es sich, über die Verwendung graphbasierter Speichermöglichkeiten nachzudenken – besonders dann, wenn man ein frisches Projekt angeht und die Tools und Frameworks noch frei wählen kann. Leider gibt es keine präzise Metrik, anhand derer man bestimmen könnte, ob eine Graphdatenbank sich wesentlich besser für das neue Projekt eignet als eine klassische SQL-Datenbank, jedoch hat sich die folgende Heuristik als recht zuverlässig erwiesen: Zeichnen Sie die Entitäten Ihrer Domäne – also die Dinge der realen Welt, die Ihre Anwendung abbilden soll – als Kreise auf ein Blatt Papier und benennen Sie diese Kreise ausschließlich mit Hauptwörtern. Verbinden Sie nun diejenigen Kreise, die miteinander in Beziehung stehen, wobei Sie auf die entstehenden Kanten ausschließlich Verben schreiben dürfen. Wenn dies gelingt und sich außerdem die Anzahl der Knoten und Kanten die Waage hält, stehen die Chancen gut, dass eine graphbasierte Speicherung für Ihre Domäne sinnvoll ist. Überlegen Sie sich nun – ausgehend von einer Entität Ihrer Wahl – einige typische Abfrageszenarien für diese Entität. Wenn der Kanten-Radius, den Sie betrachten müssen, regelmäßig zwei oder größer ist, dann könnten sich auch bei der Komplexität und Lesbarkeit der Abfragen echte Vorteile gegenüber SQL ergeben.

Es gibt andererseits auch gute Gründe, sich gegen diese spezielle Art der Datenspeicherung oder gleich ganz für den herkömmlichen SQL-Cluster zu entscheiden: Die Domäne mag so gar nicht in einen Graphen passen oder der resultierende Graph stellt sich als ausreichend simpel dar, da es weniger um Struktur als um schiere Datenmengen geht. Darüber hinaus liegt der NoSQL-Idee die (zumindest behauptete) bessere Skalierbarkeit zugrunde, welche bei verteilten Datenbanken teuer durch die Aufgabe liebgewonnener ACID-Eigenschaften erkauft werden muss (was aber entsprechend auch für die anderen NoSQL-Datenmodelle gilt). Graphdatenbanken sind daher kein Allheilmittel – es gilt: Für jedes Problem das passende Werkzeug.



Das Schema der Beispiel-Graph-datenbank (Bild 1)

Exkurs Gremlin: Über den Graphen reden

Im Hinblick auf die zu schreibende Console-App soll das verwendete Modell möglichst simpel sein: Als einzige Domänenklassen kommen Personen, Häuser, Katzen und Hunde vor. Personen kennen einander, besitzen Häuser und wohnen darin und besitzen außerdem noch Haustiere. Das Schema ist in Bild 1 grafisch dargestellt.

Wie könnte es anders sein: Auch bei Graphdatenbanken gibt es konkurrierende Vorstellungen, wie die zusammenhängenden Daten möglichst effektiv und elegant abgerufen werden sollten. Neben dem deklarativen Paradigma, dem beispielsweise Neo4j [4] mit der dazugehörigen Abfragesprache Cypher folgt [5], gibt es den prozeduralen Ansatz, den unter anderem die vom Apache-TinkerPop-Projekt [6] betreute Sprache Gremlin verfolgt.

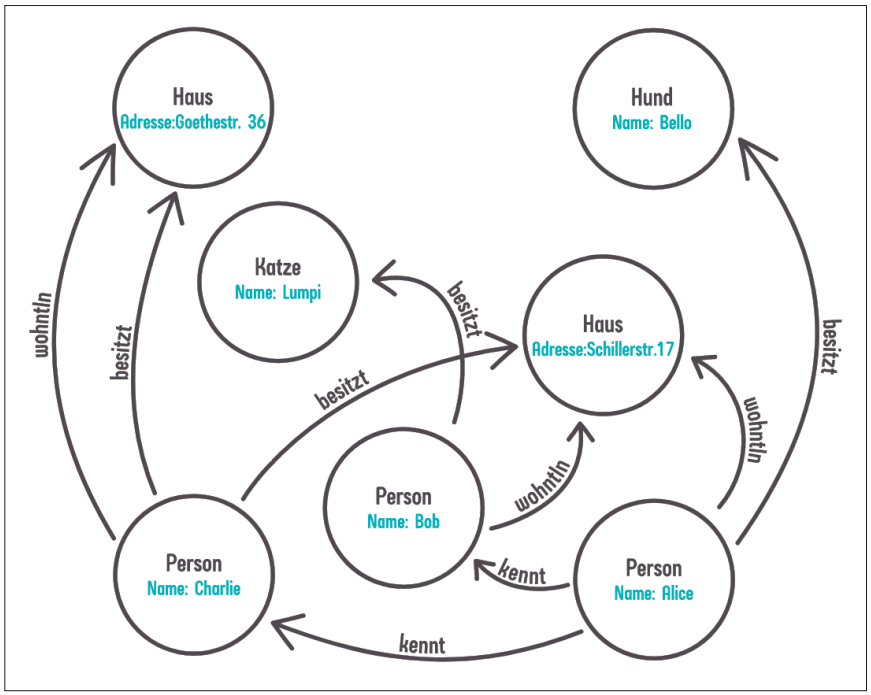
Beiden Paradigmen gemeinsam ist, dass die verwendete Sprache es erlaubt, über die Elemente eines Graphen (also die Knoten und Kanten) und die auf ihnen gespeicherten Daten zu reden. Dazu gehören neben den benutzerdefinierten Eigenschaften auch immer ein Label und eine eindeutige ID. Die Labels auf den Knoten erlauben es, diese in die verschiedenen Entitätsklassen zu unterteilen, die Sie sich beim Entwurf der Domäne überlegt haben – entsprechend dem in Bild 1 gezeigten Beispielschema würde man hier also *Person*, *Haus* et cetera verwenden. Genauso verhält es sich mit den Kanten, deren Labels mit den Verben belegt sind, die die Knoten in Beziehung zueinander setzen. Bild 2 zeigt eine auf dem genannten Schema basierende Beispieldatenbank, in der als einzige Element-Eigenschaften *Name* (für alles, was lebt) und *Adresse* (für Häuser) vorkommen.

„Prozedural" im Kontext von Gremlin bedeutet, dass Autorinnen und Autoren beim Schreiben einer Abfrage eine Traversal (wörtlich „Durchquerung") vorgeben, die explizit beschreibt, von welchen Elementen im Graphen die Reise losgeht, über welche Kanten sie in der Folge führt, wie die zu gehenden Pfade verzweigen und nach welchen Kriterien die besuchten Elemente ausgefiltert werden.

Alle in diesem kurzen Exkurs gezeigten Abfragen können einfach nachvollzogen werden (was zwar für das Verständnis nicht zwingend notwendig ist, jedoch trotzdem Spaß macht), indem Sie die Gremlin Console mit dem folgenden Befehl in einen Docker-Container laden:

```
docker run tinkerpop/gremlin-console
```

Alternativ können Sie sich die Gremlin Console von der TinkerPop-Website [6] laden und das zur jeweiligen Plattform passende Start-Skript im *bin*-Ordner ausführen, was jedoch eine korrekt installierte Java-Umgebung voraussetzt. Nach dem Start der Console geben Sie dann Folgendes ein:



Kleine, aber vollständige Graphdatenbank (Bild 2)

```
graph = TinkerGraph.open();
```

Schon haben Sie eine lokale In-Memory-Graphdatenbank. Alle Abfragen an diese Datenbank (sprich die Traversals) beginnen bei einer sogenannten Traversal Source, die mit

```
g = graph.traversal()
```

abgerufen und traditionell mit *g* bezeichnet wird. Am schnellsten zu einem Erfolgserlebnis kommen Sie mit folgender Abfrage, die den ersten mit dem Label *Person* versehenen Knoten in die Datenbank schreibt:

```
g.addV('Person').property('Name',
  'Alice')
```

Um zu überprüfen, ob Alice es auch wirklich in die Datenbank geschafft hat, können Sie nun einmal

```
g.V()
```

probieren, wodurch ungefiltert alle Knoten (Vertices*)* des Graphen aufgelistet werden. Falls Ihnen als Ergebnis die schlichte ID des einzigen Knotens zu langweilig ist, versuchen Sie es einmal mit

```
g.V().values('Name')
```

Analog dazu liefert

```
g.E()
```

ungefiltert alle Kanten (Edges) des Graphen zurück (auch wenn die Liste zum gegenwärtigen Zeitpunkt noch leer ist).

Die einzelnen durch Punkte getrennten Segmente einer Traversal nennen sich Traversal Steps, und sowohl der V-Step als auch der E-Step erlauben die Angabe numerischer oder alphanumerischer IDs, um den Beginn der Traversal auf die durch die IDs referenzierten Elemente einzuschränken.

Damit haben Sie jetzt schon alles, was Sie brauchen, um zumindest die Knoten aus der im Bild 2 gezeigten Beispieldatenbank anzulegen. Das Verbinden von Knoten durch Kanten ist ebenfalls nicht besonders kompliziert: Um zum Beispiel Bob zu einer Person zu machen, die Alice bekannt ist, führen Sie

```
g.V(0L).addE('kennt').to(__.V(2L))
```

aus, wobei einfach mal die Kenntnis der jeweiligen IDs von Alice und Bob vorausgesetzt wurde. Das Postfix *L* rührt daher, dass die IDs in der Standardeinstellung der Gremlin Console vom Java-Typ *Long* sein müssen.

Die beiden Unterstriche repräsentieren eine sogenannte anonyme Traversal, die immer dann verwendet wird, wenn ein Argument für einen Schritt wiederum eine Traversal verlangt – der Startpunkt für die innere anonyme Traversal ist dann immer das Element, an dem sich die äußere Traversal gerade befindet. Was aber, wenn Sie zwar die Knoten für Alice und Bob angelegt haben, deren IDs aber gar nicht ken- ▶

Listing 1: Konfiguration von ExRam.Gremlinq

```
private static async Task Main()
{
  var querySource = GremlinQuerySource.g
    .ConfigureEnvironment(env => env
      .UseModel(GraphModel
        .FromBaseTypes<Vertex, Edge>(lookup =>
            lookup
          .IncludeAssembliesOfBaseTypes()))
      .UseGremlinServer(builder => builder
        .AtLocalhost()));

  await new Program(querySource).Run();
}
```

nen? Dann müssen Sie die Datenbank fragen und dabei einen Filter nutzen:

```
g.V().hasLabel('Person').has('Name','Alice')
```

gibt genau den Knoten für Alice zurück, dessen ID nun in der Console angezeigt wird. Damit wäre alles gezeigt, was Sie brauchen, um auch noch die Kanten zwischen den einzelnen Knoten anzulegen – fertig ist die Beispieldatenbank! Aber wen kennt denn Alice? Jetzt bewegt sich was im Graphen:

```
g.V().hasLabel('Person').has('Name','Alice').
  out('kennt').hasLabel('Person')
```

startet vom Knoten für Alice und findet durch Überquerung aller Kanten, die das Label *kennt* tragen, alle Personen, die

Listing 2: Neuaufbau der Datenbank

```
private async Task Graph_erstellen()
{
  await g.V().Drop();

  _alice = await _g
    .AddV(new Person { Name = "Alice" })
    .FirstAsync();

  _bob = await _g
    .AddV(new Person { Name = "Bob"  })
    .FirstAsync();

  _charlie = await _g
    .AddV(new Person { Name = "Charlie" })
    .FirstAsync();

  var schillerStrasse17 = await _g
    .AddV(new Haus { Address = "Schillerstr. 17" })
    .FirstAsync();

  var goetheStrasse36 = await _g
    .AddV(new Haus { Address = "Goethestr. 36" })
    .FirstAsync();

  await _g
    .V(_charlie.Id)
    .AddE<Besitzt>()
    .To(__ => __
      .V(schillerStrasse17.Id));

  await _g
    .V(_charlie.Id)
    .AddE<Besitzt>()
    .To(__ => __
      .V(goetheStrasse36.Id));

  var bello = await _g
    .AddV(new Hund { Name = "Bello" })
    .FirstAsync();

  var lumpi = await _g
    .AddV(new Katze { Name = "Lumpi" })
    .FirstAsync();

  await _g
    .V(_alice.Id)
    .AddE<Kennt>()
    .To(__ => __
      .V(_bob.Id));

  await _g
    .V(_alice.Id)
    .AddE<Kennt>()
    .To(__ => __
      .V(_charlie.Id));

  await _g
    .V(_alice.Id)
    .AddE<Besitzt>()
    .To(__ => __
      .V(bello.Id));

  await _g
    .V(_bob.Id)
    .AddE<Besitzt>()
    .To(__ => __
      .V(lumpi.Id));

  await _g
    .V(_alice.Id)
    .AddE<WohntIn>()
    .To(__ => __
      .V(schillerStrasse17.Id));

  await _g
    .V(_bob.Id)
    .AddE<WohntIn>()
    .To(__ => __
      .V(schillerStrasse17.Id));

  await _g
    .V(_charlie.Id)
    .AddE<WohntIn>()
    .To(__ => __
      .V(goetheStrasse36.Id));
}
```

Alice kennt. Kennt umgekehrt auch Bob Alice? Bislang nicht, zumindest käme die entsprechende Abfrage nach diesem Muster nicht zu diesem Ergebnis. Es soll allerdings davon ausgegangen werden, dass dem so ist, die Traversierung muss also zwei Wege gehen: Es sind nicht nur ausgehende *kennt*-Kanten zu betrachten, sondern auch eingehende. Entsprechend könnten Sie Folgendes schreiben:

```
g.V().hasLabel('Person').has('Name','Bob').union(__.out(
  'kennt'),__.in('kennt')).hasLabel('Person')
```

Oder einfacher:

```
g.V().hasLabel('Person').has('Name','Bob').both(
  'kennt').hasLabel('Person')
```

Die vollständige Referenz zu Gremlin mit vielen weiteren Beispielen findet sich unter [7].

Gremlin + LINQ = Gremlinq

So intuitiv so eine Gremlin-Abfrage auch sein mag: Mit steigender Länge und Verschachtelungstiefe der Abfragen und speziell dann, wenn man sich am Anfang des Entwicklungsprozesses befindet, macht es wenig Spaß, solche Zeichenketten händisch zusammenzubasteln und sie über die gesamte Entwicklungszeit zu warten und manuell anzupassen.

Mit dem ebenfalls vom Apache TinkerPop-Projekt betreuten Projekt Gremlin.Net [8] kann man das Schreiben von Abfragen schon sehr vereinfachen: Als Implementierung einer sogenannten Gremlin Language Variant (GLV) portiert es die Gremlin-Interfaces nach C#, sodass alle oben gezeigten Abfragen mit wenig Aufwand in C#-Code konvertiert werden können. Allerdings befreit es einen trotzdem nicht vom Umgang mit blanken Strings für Labels und Properties. Darüber hinaus liegen die von der Datenbank zurückgelieferten Ergebnisse immer als C#-*dynamic*-Objekt vor, der User muss sich also selbst um die korrekte Interpretation der Ergebnisse kümmern. Wer aus der SQL-Welt den Komfort eines Object-Graph-Mappers (ORM) wie Entity Framework oder NHibernate gewohnt ist und damit bei der Formulierung von Abfragen nicht mehr in Tabellen, sondern in C#-Klassen denkt, will auf diese Annehmlichkeit auch im Umgang mit Graphdatenbanken oft nicht verzichten.

Angetreten, diese Funktionalität zu bieten, ist ExRam.Gremlinq [9]. Das Kunstwort aus Gremlin und LINQ (Language-Integrated Query) deutet schon darauf hin, dass ExRam.Gremlinq das Ziel verfolgt, Abfragen an Graphdatenbanken nur unter Zuhilfenahme einer C#-DSL und C#-Domänenklassen (den POCOs, für „Plain Old C-Object“) formulieren zu können. Die Abfragen werden dadurch typsicher, unterstützen Vererbung out of the box und können sehr leicht refaktorisiert werden. ExRam.Gremlinq baut auf dem schon erwähnten Gremlin.NET auf und kümmert sich darum, gültige Gremlin-Abfragen zu erzeugen und die von der Datenbank zurückgegebene Ergebnismenge wieder in POCOs oder skalare Werte zu übersetzen.

ExRam.Gremlinq ist kürzlich in der stabilen Version 8.0 erschienen und steht unter MIT-Lizenz.

Endlich Code!

Das Beispielprojekt, das Sie in den Downloads zu diesem Artikel finden, bindet das NuGet-Paket ExRam.Gremlinq.Providers.GremlinServer [10] ein, welches das Basis-Paket ExRam.Gremlinq.Core und Unterstützung für den Gremlin Server, den In-Memory-Datenbankserver des TinkerPop-Projekts, mitbringt. Durch die einzelnen ExRam.Gremlinq.Providers.*-NuGet-Pakete werden neben dem Gremlin Server auch noch einige andere Datenbanken unterstützt, wie etwa Azure Cosmos DB, ein Multi-Model-Cloud-Datenbank-Service, für den Microsoft seit März 2020 ein monatliches ▶

Listing 3: Abfrage mit Sortierung nach Namen

```
private async Task Wen_kennt_Alice()
{
  var bekannte = await _g
    .V<Person>()
    .Where(x => x.Name == "Alice")
    .Both<Kennt>()
    .OfType<Person>()
    .Order(x => x
       .By(x => x.Name));

  foreach (var person in bekannte)
  {
    Console.WriteLine($"Alice kennt {
      person.Name}.");
  }
}
```

Listing 4: Abfrage nach Namens-Anfangsbuchstabe B

```
private async Task Wer_besitzt_ein_Haustier_dessen_
    Name_mit_B_anfängt()
{
  var besitzer = await _g
    .V<Person>()
    .Where(__ => __
      .Out<Besitzt>()
      .OfType<Haustier>()
      .Where(x => x.Name.StartsWith("B")));

  foreach (var person in besitzer)
  {
    Console.WriteLine($" {person.Name} besitzt ein
      Haustier, dessen Name mit 'B' anfängt.");
  }
}
```

Gratis-Kontingent anbietet [11] und der sich dadurch sehr für kleinere Projekte ohne große Ansprüche an Speicherplatz und Skalierbarkeit empfiehlt.

Soll der oben genannte Gremlin-Server wie vorgegeben Verwendung finden, muss dieser selbstverständlich noch ans Laufen gebracht werden. Am einfachsten geht das wieder mit einem Docker-Container:

```
docker run -p 8182:8182 tinkerpop/gremlin-server
```

Alternativ lässt sich die aktuelle Version des Servers als ZIP-Archiv von der Apache TinkerPop-Website [7] laden und entpacken. Genau wie bei der oben schon erwähnten Gremlin Console ist nun je nach Betriebssystem das entsprechende Skript im *\bin*-Ordner auszuführen. Auch hierfür ist eine korrekt konfigurierte Java-Umgebung erforderlich.

Bevor es möglich ist, die erste Abfrage an die Datenbank zu stellen, muss ExRam.Gremlinq konfiguriert werden. Während die Konfiguration im tatsächlichen Projekt etwas ausführlicher ausfällt, reicht der in Listing 1 gezeigte Code im Grunde dazu völlig aus. Schon hier zeigt sich der funktionale Stil, der sich fortan auch durch die weitere Verwendung von ExRam.Gremlinq ziehen wird: Objekte aus dem ExRam.Gremlinq-Framework sind unveränderbar, stattdessen werden die auf den Klassen verfügbaren Methoden im Fluent-Interface-Stil so verknüpft, dass sich immer neue Objekte ergeben. Der Code in Listing 1 zeigt, wie auf diese Art sowohl die C#-Klassen *Knoten* und *Kante* als Basisklassen definiert werden, als auch die Netzwerkadresse des Gremlin Servers, zu dem eine Verbindung aufgebaut werden soll, festgelegt wird.

Ist die Konfiguration abgeschlossen, so erhält man ein Objekt vom Typ *IGremlinQuerySource*, das im weiteren Ablauf des Programmcodes der Dreh- und Angelpunkt für alle Datenbankabfragen ist.

Jetzt geht es endlich los: Listing 2 zeigt, wie zu Beginn der Ausführung die verbundene Datenbank erst einmal vollständig gelöscht wird, um sie dann Schritt für Schritt so aufzubauen, wie es Bild 2 zeigt.

Bis auf die für C# üblichen großgeschriebenen Methodennamen und einige *await*-Statements sieht der Code für ExRam.Gremlinq fast so aus wie pures Gremlin, zumindest sind die Entsprechungen sofort sichtbar. Aber auch Unterschiede werden sofort deutlich: Statt jede Eigenschaft eines neuen Knotens dediziert zu spezifizieren, erzeugen Sie ein Objekt, setzen dessen Eigenschaften und geben dieses Objekt dann in die *AddV*-Methode. Genauso verhält es sich mit den Ergebnissen: Auch hier kommen mit allen Eigenschaftswerten bestückte Objekte zurück.

Ein weiteres Feature von ExRam.Gremlinq ist das intelligente Typ-System, welches das Fluent Interface ausmacht: ExRam.Gremlinq unterscheidet sehr genau, ob eine Query als Ergebnismenge Knoten, Kanten, Properties oder skalare Werte liefern wird. Dadurch kann es bei jedem Schritt genau die Methoden anbieten, die gerade im Kontext der Query sinnvoll sind.

Mit der Abfrage, wen Alice kennt, befasst sich Listing 3: Im Prinzip ist sie ja schon ein alter Hut. Um sie also etwas aufzumotzen, sei noch eine Sortierung nach dem Namen eingefügt. Hierbei wird nun zum ersten Mal ein größerer Unterschied in der Namensgebung zwischen Gremlin und ExRam.Gremlinq deutlich. Während in purem Gremlin noch mit dem *hasLabel*-Schritt gearbeitet wurde, um nur Knoten mit dem Label *Person* zu betrachten, lehnt sich ExRam.Gremlinq hier mehr an die bekannten LINQ-Operatoren an. Da auf dieser Ebene gar nicht mehr von Labels die Rede ist, sondern nur von Domänenklassen, wird aus *hasLabel('Person')* nun *OfType<Person>()* beziehungsweise aus *V().hasLabel('Person')* ganz einfach *V<Person>()*. Außerdem wird aus *has('Name', 'Alice')* ganz LINQ-typisch *Where(x => x.Name == "Alice")*.

Listing 5: Abfrage nach Name und Anzahl

```
private async Task Wer_besitzt_wie viele_Haustiere()
{
  var tuples = await _g
    .V<Person>()
    .Project(p => p
      .ToDynamic()
      .By("name", __ => __
        .Values(x => x.Name))
      .By("anzahl", __ => __
        .Out<Besitzt>()
        .OfType<Haustier>()
        .Count()));

  foreach (var tuple in tuples)
  {
    Console.WriteLine($" {tuple.name} besitzt {
      tuple.anzahl} Haustiere.");
  }
}
```

Listing 4 behandelt die – zugegebenermaßen etwas konstruierte – Frage, wer ein Haustier besitzt, dessen Name mit „B“ anfängt. Der entsprechende Abfragecode zeigt die Fähigkeit von ExRam.Gremlinq auf, mit einer großen Range an C#-Expressions umgehen zu können, die sich in valides Gremlin übersetzen lassen, in diesem Fall die Expression *x => x.Name.StartsWith("B")*. Die hierzu passende Gremlin-Abfrage würde in etwa so aussehen:

```
g.V().hasLabel('Person').where(__.out('Besitzt').
  hasLabel('Hund','Katze').has('Name',startingWith('B')))
```

Wie man leicht sehen kann, hat ExRam.Gremlinq die Einschränkung auf den abstrakten Typ *Haustier* in eine Einschränkung auf die Labels *Hund* und *Katze* übersetzt – denn im Graph hat kein Knoten das Label *Haustier*. ExRam.Gremlinq kennt also, dank der eingangs vorgenommenen Konfiguration eines Graph-Models, die gesamte Typenhierarchie der verwendeten Entitätstypen und formuliert die Abfrage entsprechend korrekt.

Reines Gremlin beziehungsweise ExRam.Gremlinq liefert aber nicht nur ganze Knoten, Kanten oder Skalare, sondern ist auch in der Lage, durch Projektion eigene Strukturen zu erzeugen. Das wird immer dann notwendig, wenn man für ein Ausgangselement (zum Beispiel den Knoten für Alice) exemplarisch sowohl den Name-Eigenschaftswert abfragen will als auch die Anzahl der Haustiere, die Alice besitzt. Ganz abstrakt behandelt Listing 5 die Frage „Wie viele Haustiere hat denn jeder?“, und als Ergebnis wird eine Liste von C#-Dynamics mit den Eigenschaften *name* und *anzahl* erwartet.

Zu guter Letzt soll noch die Frage beantwortet werden, wer mit wem zusammenwohnt. Im Unterschied zum vorherigen Beispiel soll sich das Ergebnis diesmal als eine Liste von C#-ValueTuples darstellen.

Die Traversal geht also wieder von allen Personen aus und projiziert dann sowohl auf die Person selbst als auch auf die Liste ihrer Mitbewohnerinnen und Mitbewohner. Als Mitbewohnerin oder Mitbewohner soll einfachheitshalber eine Person gelten, die im selben Haus wohnt. Das ist schnell als Traversal formuliert: die *WohntIn*-Kante ausgehend traversieren und daraufhin über alle eingehenden *WohntIn*-Kanten zurückgehen. Leider findet die Traversal damit aber auch wieder die Person, von der die Reise ausging. Es muss also gelingen, sich die Person, für die sich diese Frage stellt, zu merken, um sie später herauszufiltern. Genau das leistet der *As*-Step, der es erlaubt, dem aktuell betrachteten Element einen Namen zu geben, auf den dann im weiteren Verlauf wieder Bezug genommen werden kann. Wie das Ganze in ExRam.Gremlinq aussieht, zeigt Listing 6.

Nahezu die gesamte expressive Kraft von Gremlin wird durch ExRam.Gremlinq elegant abgebildet, sodass sich die entstehenden Abfragen nicht nur gut schreiben und warten beziehungsweise refaktorisieren, sondern auch miteinander kombinieren lassen. Die Erzeugung ganzer domänenspezifischer Sprachen (DSLs) ist hier denkbar, würde aber den Rahmen dieses Artikels sprengen.

Listing 6: Abfrage nach Mitbewohnern

```
private async Task Wer_wohnt_mit_wem_zusammen()
{
  var mitbewohnerListe = await _g
    .V<Person>()
    .As((__, person) => __
       .Project(p => p
         .ToTuple()
         .By(__ => __
           .Values(x => x.Name))
         .By(__ => __
           .Out<WohntIn>()
           .In<WohntIn>()
           .OfType<Person>()
           .Where(anderePerson => anderePerson
             != person.Value)
           .Values(x => x.Name)
           .Fold())));

  foreach (var mitbewohner in mitbewohnerListe)
  {
    if (mitbewohner.Item2.Any())
      Console.WriteLine($" {mitbewohner.Item1}
        wohnt mit {string.Join(',',
        mitbewohner.Item2)} zusammen.");
    else
      Console.WriteLine($" {mitbewohner.Item1}
        wohnt mit niemandem zusammen.");
  }
}
```

Fazit

Graphdatenbanken scheinen, gerade was ihre Verwendung in .NET-Backends betrifft, immer noch in einer Art Nische zu existieren – zu Unrecht, wie dieser Artikel hoffentlich gezeigt hat. Der Einstieg in die Abfragesprache Gremlin ist einfach, und Cloud-Provider, die gehostete Graphdatenbanken anbieten, gibt es mittlerweile einige. Tools wie das vorgestellte ExRam.Gremlinq treten an, den Komfort der aus der SQL-Welt bekannten ORMs auch auf Graphdatenbanken zu übertragen. Den Quellcode der Beispiele finden Sie in den Downloads zum Artikel. ■

[1] Azure Cosmos DB, www.dotnetpro.de/SL2102ExRamGremlinq1
[2] JanusGraph, www.dotnetpro.de/SL2102ExRamGremlinq2
[3] Neptune, https://aws.amazon.com/de/neptune
[4] Neo4j, https://neo4j.com
[5] Cypher Query Language, www.dotnetpro.de/SL2102ExRamGremlinq3
[6] Apache TinkerPop, https://tinkerpop.apache.org
[7] Referenz zu Gremlin, www.dotnetpro.de/SL2102ExRamGremlinq4
[8] Gremlin.Net for Apache TinkerPop, www.dotnetpro.de/SL2102ExRamGremlinq5
[9] ExRam.Gremlinq, www.dotnetpro.de/SL2102ExRamGremlinq6
[10] ExRam.Gremlinq.Providers.GremlinServer, www.dotnetpro.de/SL2102ExRamGremlinq7
[11] Gratiskontingent in Azure Cosmos DB, www.dotnetpro.de/SL2102ExRamGremlinq8

Daniel Weber
studierte Informatik und entwickelt seit 2002 für die .NET-Plattform. Seine Schwerpunkte liegen im Bereich Graphdatenbanken und asynchrone/reaktive Programmierung unter .NET. Zu erreichen unter @danielcweber oder **me@danielcweber.net**.

dnpCode A2102ExRamGremlinq