Microsoft

Azure Cosmos DB 自習書

Azure Cosmos DB Gremlin API 編

この自習書では、Azure Cosmos DB Gremlin API を使用して、エンティティとリレーションシップの両方をモデル化できるグラフデータベースの構築と操作の方法を学習します。

発行日：2019 年 1 月 16 日

Edifist
エディフィストラーニング株式会社

Microsoft Partner
Microsoft
Gold Learning
Gold Data Platform
Silver Data Analytics

更新履歴

版数	発行日	更新履歴
第 1 版	2019 年 1 月 16 日	初版発行

本書の取り扱い注意事項

本書は、2019 年 1 月時点の Azure Cosmos DB をベースに作成しています。読者が本書を読まれる時点では、Azure ポータルの UI や提供される機能などが変更されている可能性があることをあらかじめご了承ください。

なお、Azure サービスの更新情報に関しては、「Azure の更新情報」を参照してください。

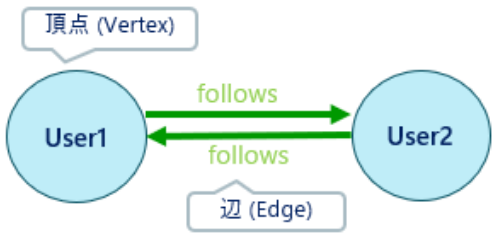
目次

1. はじめに

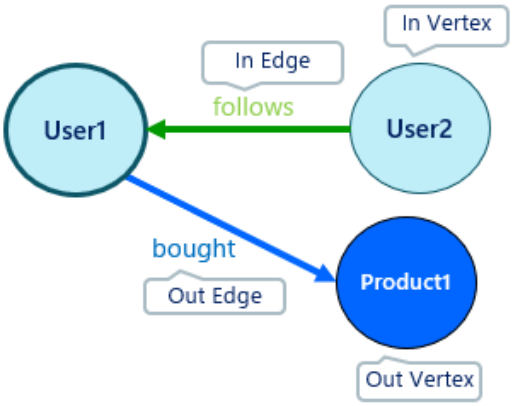
Azure Cosmos DB は、Microsoft Azure PaaS (Platform as a Services) 環境で提供されている NoSQL で利用可能なデータベース サービスです。Azure Cosmos DB の特徴は、キー/バリュー ストア、ドキュメント データベース、グラフ データベースといったマルチ データモデルに対応し、選択したリージョンに即座にデプロイできるグローバル分散型のサービスで、予測可能な応答性能を提供できるように設計されていることです。

この自習書では、オープンソースの Apache TinkerPop Gremlin に準拠する Azure Cosmos DB のグラフ データベースの使用方法を学習します。グラフ データベースのグラフ (Graph) とは、折線グラフや棒グラフのようなグラフのことではなく、情報と情報をつなぐ関係性を表現するデータ構造 (ネットワーク) を意味します。リレーションシップ自体をエンティティとして格納できるため、データそのものより、データ同士のリレーションシップが主な関心事で、複雑に絡み合ったデータの関係性を迅速にクエリする必要がある場合に有効に機能します。

グラフ データベースでは、データ エンティティを頂点 (Vertex) に格納し、データ エンティティ間のリレーションシップを辺 (Edge) に格納します。



Azure Cosmos DB のグラフ構造の Edge は、単純な結線ではなく、方向を持つため、有向グラフと呼ばれています。下図では、follows は、Vertex User1 に入ってくる Edge (In Edge) で、bought は、Vertex User1 から出ていく Edge (Out Edge) であることを説明しています。有向グラフにより、データ エンティティ間の親子関係や経路などを表すことができます。



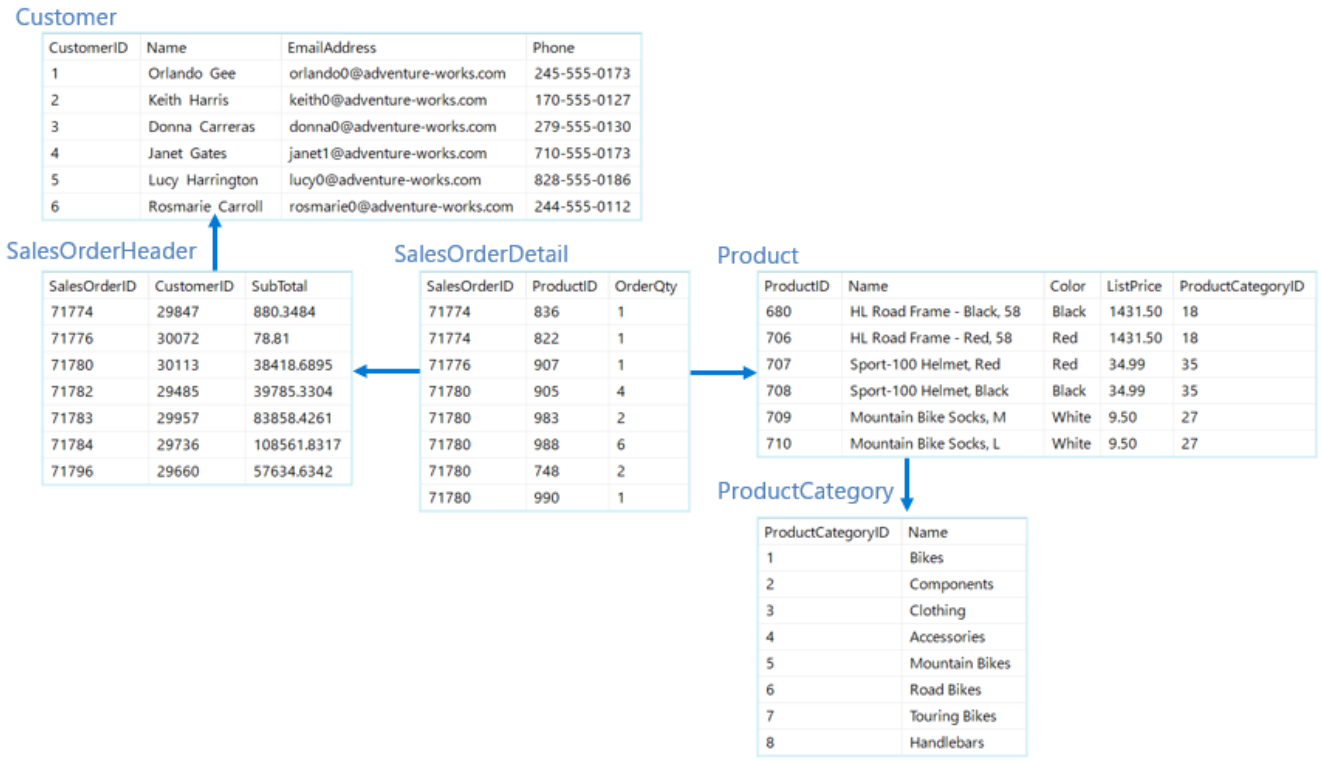
この自習書では、本章でグラフ データベースの特性や必要性を説明した後、第 2 章で、Azure ポータルからグラフ データベースとコンテナーを作成した後、コンテナーにグラフ構造の Vertex と、Edge を格納しグラフの構造を構築します。また、Azure ポータルの Data Explorer を使用して、グラフ構造に対する検索操作を試してみます。

第 3 章では C# で Gremlin API を使用するアプリケーション コードを記述し、Azure ポータルから行った操作をアプリケーション コードから実行します。なお、本書に掲載したサンプル コードは、Windows 10 で実行することを前提とします。

1.1 グラフ データベースの必要性

どのような時にリレーショナル データベースではなく、グラフ データベースを使用するべきでしょう？

次のダイアグラムは、AdventureWorks sample databases で公開されている adventureworkslt サンプル データベースのテーブルの一部を掲載しています。



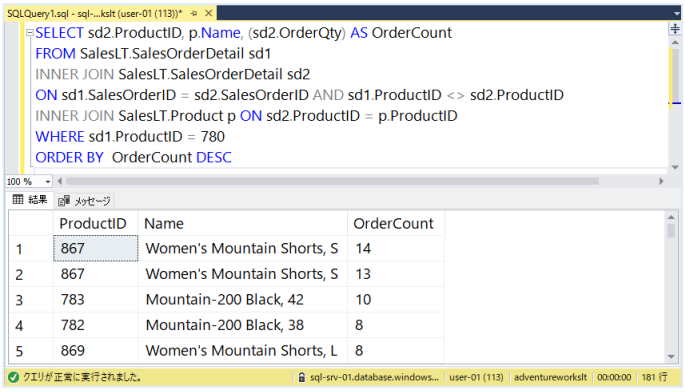
これらのテーブルは、注文処理を実行するアプリケーションのために設計され SalesOrderHeader と SalesOrderDetail テーブルに注文データを挿入し、注文された商品の検索などの一般的な受注処理の要求を満たすことができます。

しかし、昨今のビジネス要求を考えてみると、これらのテーブルに格納されたデータから、「ある商品を購入しようとしている顧客に対し、その商品を購入した他の顧客は、他にどの商品を購入したのか？」とか「ある商品は、どの商品と一緒に購入されるか？」といった情報を導き出すことも要求されますが、これらのテーブル構造で対応しようとすると困難な状況に直面します。

OLTP 処理を目的としたテーブル構造に対して、分析系の要求を満たすクエリを実装しようとすると、コストの高いクエリを実行しなければならなくなります。次のサンプル クエリは、ProductID 780 の Mountain-200 Silver, 42 (マウンテン バイク) を購入するとき、一緒に購入された商品を数の多い順序でリストします。

```
SELECT sd2.ProductID, p.Name, (sd2.OrderQty) AS OrderCount
FROM SalesLT.SalesOrderDetail sd1
INNER JOIN SalesLT.SalesOrderDetail sd2
ON sd1.SalesOrderID = sd2.SalesOrderID AND sd1.ProductID <> sd2.ProductID
INNER JOIN SalesLT.Product p ON sd2.ProductID = p.ProductID
WHERE sd1.ProductID = 780
ORDER BY OrderCount DESC
```

2. Azure ポータルからのデータベースの作成と操作



```
SELECT sd2.ProductID, p.Name, (sd2.OrderQty) AS OrderCount
FROM SalesLT.SalesOrderDetail sd1
INNER JOIN SalesLT.SalesOrderDetail sd2
ON sd1.SalesOrderID = sd2.SalesOrderID AND sd1.ProductID <> sd2.ProductID
INNER JOIN SalesLT.Product p ON sd2.ProductID = p.ProductID
WHERE sd1.ProductID = 780
ORDER BY OrderCount DESC
```

	ProductID	Name	OrderCount
1	867	Women's Mountain Shorts, S	14
2	867	Women's Mountain Shorts, S	13
3	783	Mountain-200 Black, 42	10
4	782	Mountain-200 Black, 38	8
5	869	Women's Mountain Shorts, L	8

このクエリは、別名を付けた 2 つの SalesOrderDetail テーブルを INNER JOIN で結合しています。結合条件として SalesOrderID が同じ (同じ注文) で異なる ProdutID (異なる商品) を指定することで、一回の注文で購入された商品の組み合わせを作成しています。

SalesOrderDetail テーブルの結合になるため、時間が経過し、テーブル サイズが大きくなるとパフォーマンスは低下するでしょう。一般的には、このようなバスケット分析処理は、別途データ ウェアハウスを作成し、受注データをファクト テーブルにロードして対応しますが、データの増加に対しては最適化を続ける必要があります。

また、リレーショナル データベースは、行と列によって構成されるテーブル (表形式のエンティティ) をリレーションシップにより関連付ける関係モデルを表現するためのデータベースですが、その名称に反し、エンティティ間の関係性にフォーカスしたデータ モデルを扱うことには向いていません。

リレーショナル データベースでの「リレーションシップ」は、テーブル間の整合性を保つためだけに存在しているため、結合の強度や重み、その他属性情報を一緒に管理したいと考えた場合、テーブル間の結合を管理する複数の外部キーを持った多対多のリンク テーブルを作成する必要があります。

例えば、Adventureworks 社が、自社の商品を購入した顧客が参加できる掲示板サイトやコミュニティを用意し、顧客同士の交流や商品の新しい活用方法といった情報を提供したいと考えた場合、顧客同士の関係性を格納するために次のような多対多のテーブルを追加する必要があります。

Customer (c1)

CustomerID	Name
1	Orlando Gee
2	Keith Harris
3	Donna Carreras
4	Janet Gates
5	Lucy Harrington
6	Rosmarie Carroll

Follower (f)

CustomerID	FollowerID
1	2
1	6
2	1
2	3
2	5
6	4

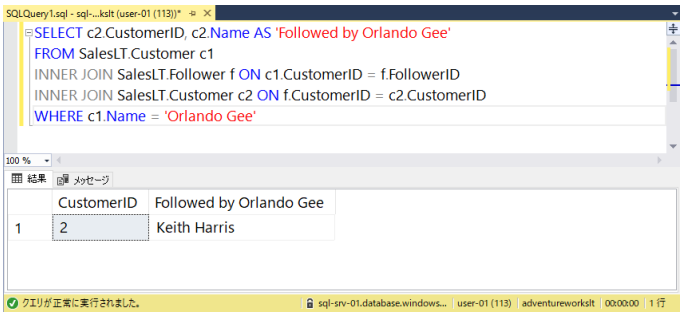
Customer (c2)

CustomerID	Name
1	Orlando Gee
2	Keith Harris
3	Donna Carreras
4	Janet Gates
5	Lucy Harrington
6	Rosmarie Carroll

Orlando Gee (CustomerID = 1) が、フォローしている顧客の検索で考えてみましょう。この場合、これらのテーブルに対して、次のクエリを実行します。

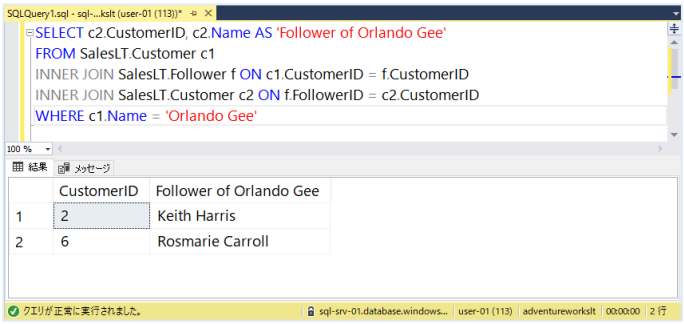
```
SELECT c2.CustomerID, c2.Name AS 'Followed by Orlando Gee'
FROM SalesLT.Customer c1
INNER JOIN SalesLT.Follower f ON c1.CustomerID = f.FollowerID
INNER JOIN SalesLT.Customer c2 ON f.CustomerID = c2.CustomerID
WHERE c1.Name = 'Orlando Gee'
```

2. Azure ポータルからのデータベースの作成と操作



このクエリの結果から、Orlando Gee は、Keith Harris (CustomerID = 2) をフォローしていることが分かるでしょう。また、逆に Orlando Gee をフォローしている顧客を検索する場合は、次のクエリを実行します。

```
SELECT c2.CustomerID, c2.Name AS 'Follower of Orlando Gee'
FROM SalesLT.Customer c1
INNER JOIN SalesLT.Follower f ON c1.CustomerID = f.CustomerID
INNER JOIN SalesLT.Customer c2 ON f.FollowerID = c2.CustomerID
WHERE c1.Name = 'Orlando Gee'
```



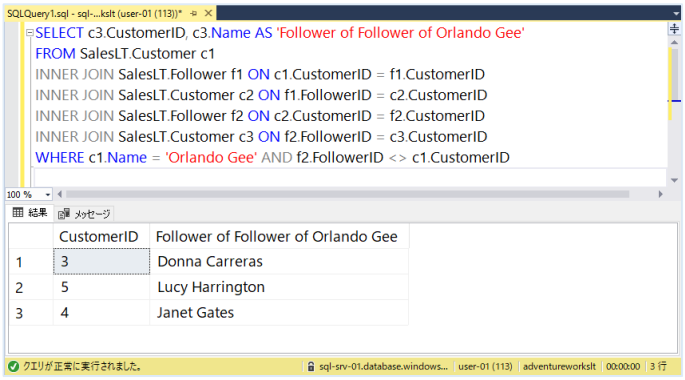
このクエリの結果から、Orlando Gee をフォローしているのは、Keith Harris (CustomerID = 2) と Rosmarie Carroll (CustomerID = 6) であることがわかるでしょう。

これらのクエリの実行では、Customer テーブルへのアクセスは、WHERE 句が機能してシーク操作になりますが、Follower テーブルには全件スキャンが必要になります。このため、顧客数が増えて、テーブルのサイズが大きくなるとパフォーマンスに影響を及ぼし始めます。さらに例えば顧客の影響範囲を確認する目的で、フォロワーのフォロワーを抽出するようなシナリオでは、使用する中間テーブルが増えていきます。

2. Azure ポータルからのデータベースの作成と操作

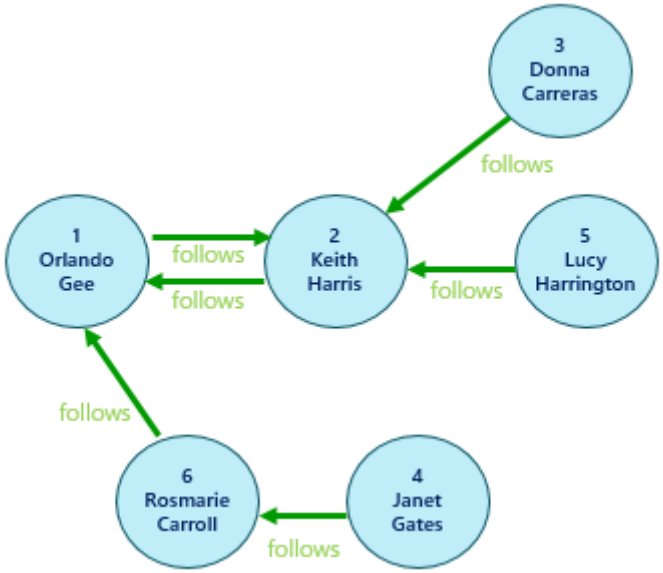
Orlando Gee をフォローしている顧客をフォローしている顧客の検索では、次のクエリを実行します。

```
SELECT c3.CustomerID, c3.Name AS 'Follower of Follower of Orlando Gee'
FROM SalesLT.Customer c1
INNER JOIN SalesLT.Follower f1 ON c1.CustomerID = f1.CustomerID
INNER JOIN SalesLT.Customer c2 ON f1.FollowerID = c2.CustomerID
INNER JOIN SalesLT.Follower f2 ON c2.CustomerID = f2.CustomerID
INNER JOIN SalesLT.Customer c3 ON f2.FollowerID = c3.CustomerID
WHERE c1.Name = 'Orlando Gee' AND f2.FollowerID <> c1.CustomerID
```



ソーシャル ネットワークの管理をリレーショナル データベースで実現しようとすると、コストの大きな結合やネストされたクエリの実行が必要となり、時間の経過とともにテーブルのサイズが増加すると、インデックス メンテナンスや並列化などの最適化をしなければならず、データベース管理に多くの労力が必要になります。また、クエリから参照する中間テーブルの数が増えていくと、パフォーマンスは飛躍的に遅くなります。

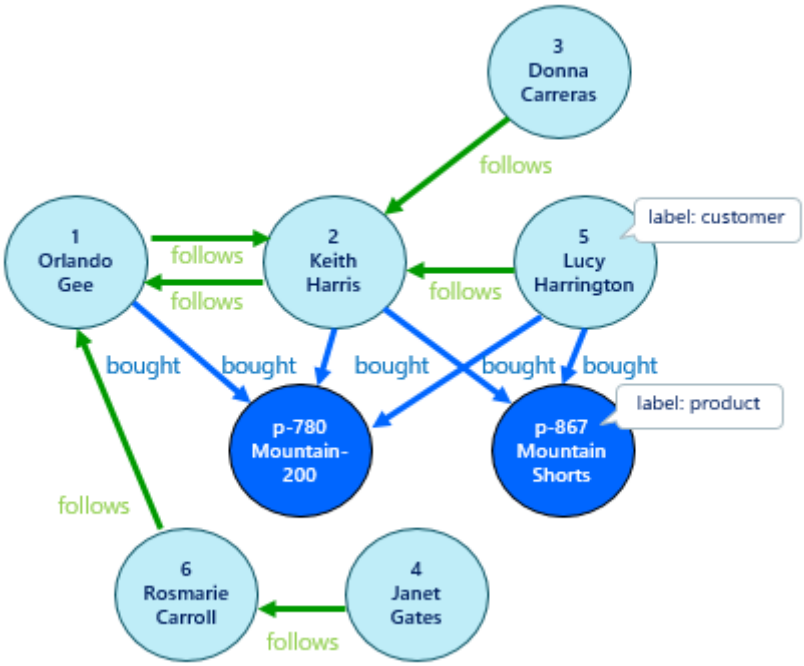
ここまでのシナリオで使用した Customer テーブルと Follower テーブルをベースに Azure Cosmos DB のグラフ データベースに移行することを考えてみましょう。グラフ データベースのコンテナーに Vertex と Edge からなる次のようなグラフ データ構造を作成します。



リレーショナル データベースに比べて直感的に理解できるわかりやすいデータ モデルを作成することができます。

2. Azure ポータルからのデータベースの作成と操作

Vertex や Edge は、ラベルを設定することで異なる種類のエンティティを保存できますので、顧客により形成されたソーシャルネットワークのグラフ構造に、商品の Vertex と購入を示す Edge を追加することができます。

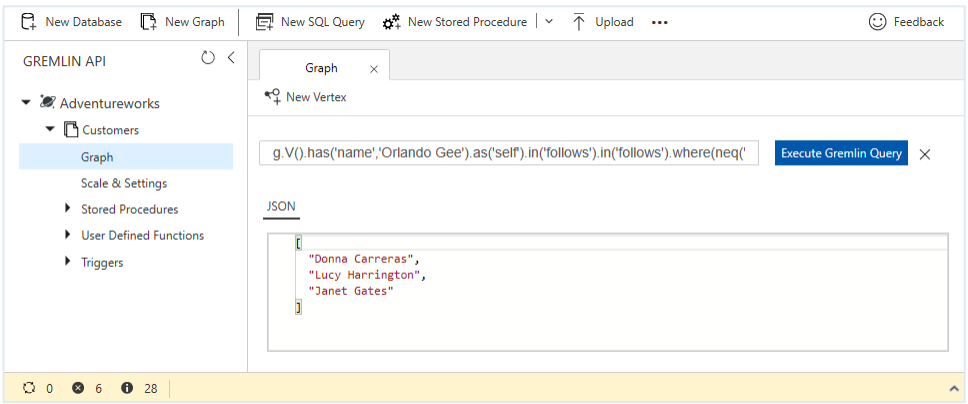


グラフ データから「Mountain-200」を購入している顧客の 3 分の 2 は、「Mountain Shorts」を購入していることがわかります。

このグラフ データ構造に対して実行するクエリは、SQL ではなくグラフ クエリ言語を使用します。この言語は、ステップからなる関数型のプログラミング言語で SQL よりもシンプルな記述で、エンティティ間の関係性を高速にトラバーサルできます。「トラバーサル」とは、「横断する」といった意味の英語で木構造やグラフ構造の全てのノードを辿ることを意味します。

例えば、上記のグラフ データ構造に対して、Orlando Gee をフォローしている顧客をフォローしている顧客の検索では、次のクエリを記述します。

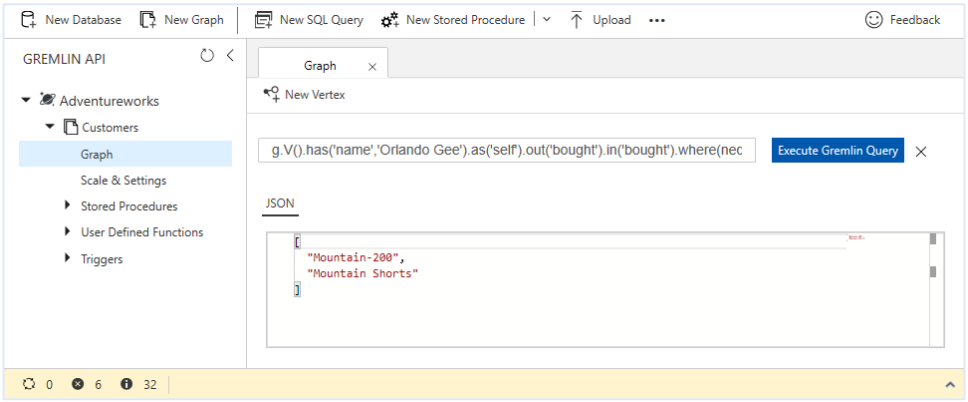
```
g.V().has('name','Orlando Gee').as('self').in('follows').in('follows').where(neq('self')).values('name')
```



このクエリは、前のページで、複数の中間テーブルを使用して実行した SQL クエリに相当します。

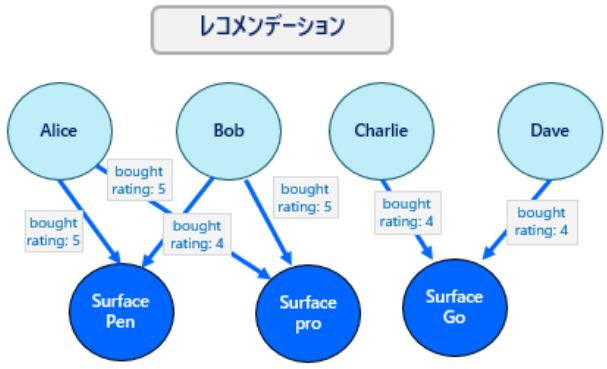
また、グラフ クエリ言語で次のように記述すると Orlando Gee が購入した商品と同じものを購入している顧客が、他にどのような商品を購入しているかを調べることができます。

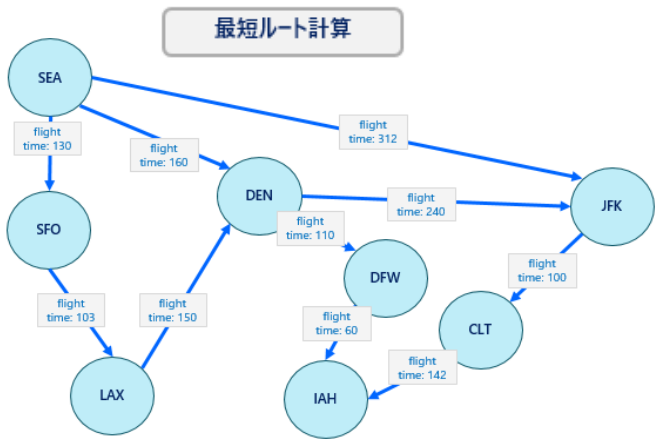
```
g.V().has('name','Orlando Gee').as('self').
      out('bought').
      in('bought').where(neq('self')).
      out('bought').
      dedup().values('name')
```



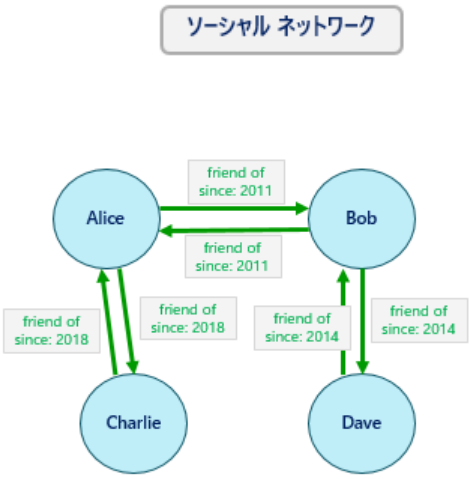
in() と out() の記述でトラバーサルの方向を設定していることがわかりますでしょうか？まだ、よくわからなくても大丈夫です。本自習書を通じて、これらのクエリ言語の基本的な使用方法をマスターすることができます。

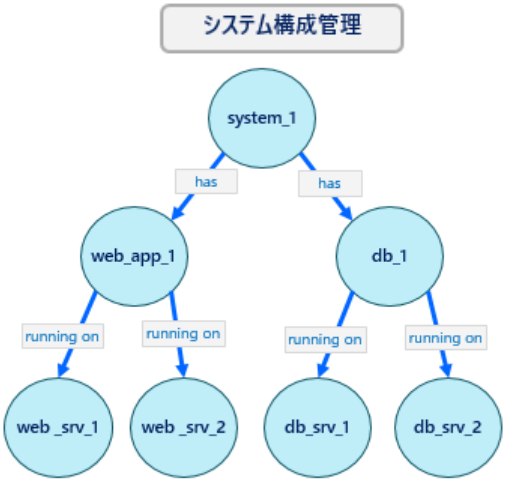
現実の世界では、リレーショナルデータベースの ER 図として抽象化される世界をはるかに超えて、あらゆる人やモノが相互に関係性をもって存在しています。ソーシャル ネットワーキング、レコメンデーション、経路検索、システム構成管理など、人と人、人とモノ、またはモノとモノの結びつきを理解する必要があるシナリオでは、リレーショナル データベースよりグラフ データベースに利点があります。





2. Azure ポータルからのデータベースの作成と操作





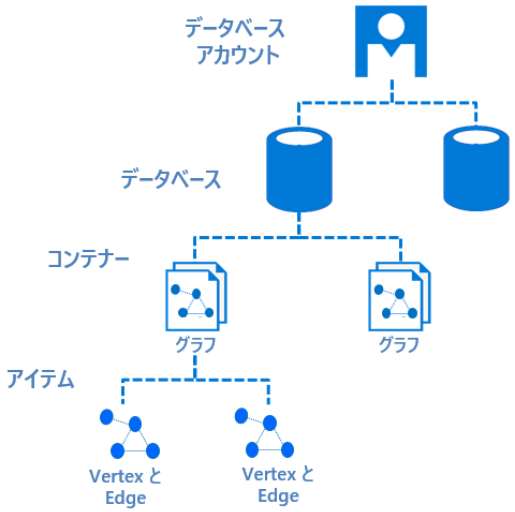
以降では実際に、Azure ポータルから Azure Cosmos DB グラフ データベースとコンテナーを作成し、コンテナーにグラフ構造を構築した後、Azure ポータルの Data Explorere から Gremlin グラフ クエリ言語を実行して、検索操作を試してみます。この言語の使用方法がわかってくると、グラフ データからいろいろな情報を取得できるようになるでしょう。

1.2 Azure Cosmos DB のグラフ データベース

Azure Cosmos DB データベース アカウントの作成で、「**Gremlin API**」を選択するとグラフ データベースが作成されます。このグラフ データベースは、Gremlin と呼ばれる Apache Tinkerpop グラフ クエリ言語を使用することができる PaaS データサービスですが、併せて Azure Cosmos DB の特徴である低テイテンシ、高スループットなデータストアを利用することができ、地理レプリケーションやマルチ マスター構成などの機能も利用できます。

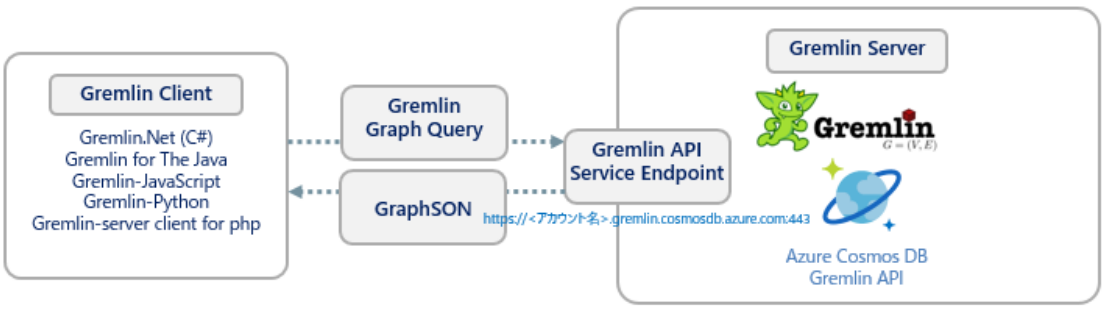
なお、データベース アカウントのデプロイ時に選択した API は、後から変更することができないことに注意してください。この手順は「2.2.1 Azure ポータルからの Azure Cosmos DB データベース アカウントの作成」で実施します。

Azure Cosmos DB グラフ データベースのリソース モデルは、下図のようにデータベース アカウントに従属する階層構造になっています。



グラフは、Vertex と Vertex 間の連結関係を表す Edge で構成される抽象データ型で、Vertex と Edge は、いずれも任意の数のプロパティを持つことができ、データの問題を直感的に扱うことができます。Azure Cosmos DB では既定の構成で、グラフの Vertex と Edge のすべてのプロパティに対して、インデックスが自動作成されるためセカンダリ インデックスの作成は不要です。

Azure ポータル、Azure CLI、および、Azure PowerShell を使用して Azure Cosmos DB Gremlin API アカウントを作成し、アカウントにアクセスすることができます。アカウントを作成したら、.NET、Java、Node.js などのプログラミング言語で作成した Gremlin クライアントから Gremlin の WebSocket フロントエンドを提供する Gremlin API サービス エンドポイント (https://<アカウント名>.gremlin.cosmosdb.azure.com:443) に接続することができ、そのアカウント内のグラフ データベースにアクセスできます。

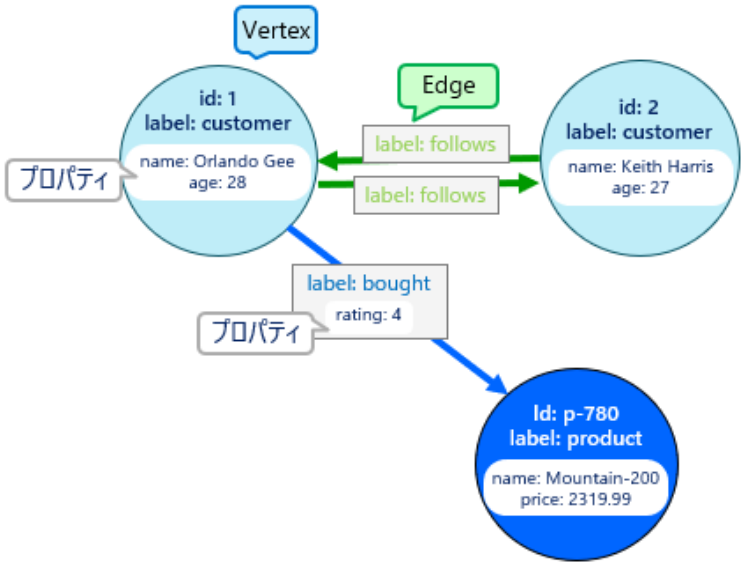


2. Azure ポータルからのデータベースの作成と操作

Azure Cosmos DB グラフ データベースを使用して、グラフの構造を構築する場合、下表の用語を正しく理解しておく必要があります。

用語	説明
Vertex	ノードとも呼ばれ、人、モノ、場所、イベントなどのデータ エンティティを表します。リレーショナル データベースのテーブルに相当します。
Edge	データ エンティティ間の関係性を表し、方向とタイプを有します。これにより、親子関係、アクション、経路、商品などの購入、推奨などを表すことができます。リレーショナル データベースのリレーションシップに相当します。
プロパティ (property)	Vertex と Edge の属性情報で、データはキー/バリュー形式で保持されます。プロパティのデータ型として String、Boolean、Number を選択できます。Vertex のプロパティは、リレーショナル テーブルの列に相当します。
id	Vertex、Edge、プロパティを識別するための一意識別子 パーティション分割されている場合、パーティション内で一意である必要があります。パーティション分割されていない場合は、コンテナーで一意である必要があります。
ラベル (label)	Vertex、Edge のタイプを示します。Vertex のラベルは、リレーショナル テーブルのテーブル名に相当します。

下図の様に Vertex を Edge で結ぶことで、Vertex 間の関係を表現できます。Edge はリレーショナル データベースのリレーションシップに似ていますが、Edge にもプロパティを記述できるため、これにより、Vertex と Edge で表現される複雑なデータ構造を表現することができます。階層型のデータ構造では扱いが困難になる循環型のデータも扱うことができます。Vertex に対する評価、コスト、時間、などを Edge のプロパティに保持できるため、最適なルートの計算やコストを計算するシナリオにも適用できます。



2. Azure ポータルからのデータベースの作成と操作

この章では、Azure ポータルを使用して、実際に Azure Cosmos DB にグラフ データベースをデプロイし、シンプルな構造の Vertex と Edge のデータを格納して、Azure ポータルの [Data Explorer] ブレードでデータを検索する手順を試してみましょう。

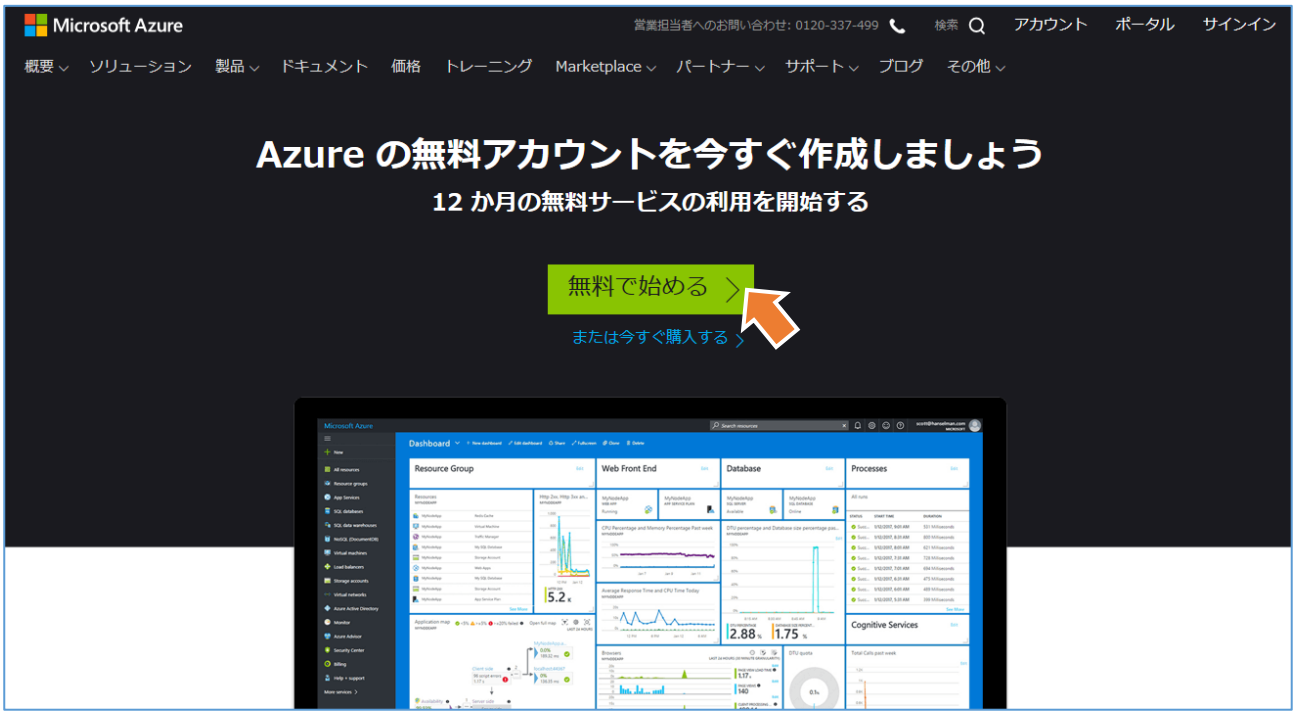
2.1 自習環境の準備

2.1.1 Microsoft Azure 無料評価版のサインアップ手順

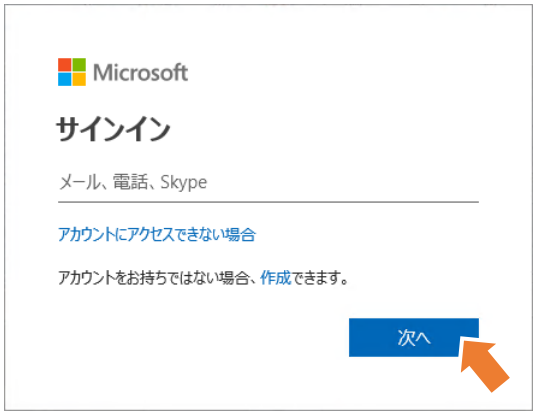
この手順はオプションです。Microsoft Azure サブスクリプションをお持ちでない場合、次の手順を実行し、Microsoft Azure の 12 ヶ月間の無料評価版サブスクリプションを取得してください。なお、この手順を完了するには、下記が必要となりますので予めご用意ください。

- 確認コードを音声または、ショートメッセージ (SMS) で受け取るための電話番号
- 身元確認のためのクレジットカード

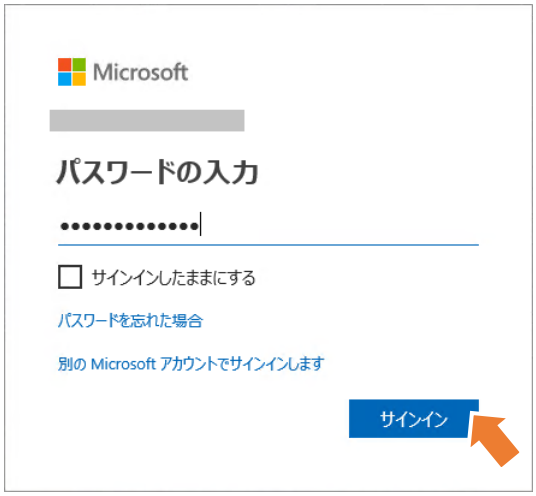
1. Web ブラウザーを起動し、https://azure.microsoft.com/ja-jp/free にアクセスします。
2. [Azure の無料アカウントを今すぐ作成しましょう] が表示されるので、[無料で始める] をクリックします。



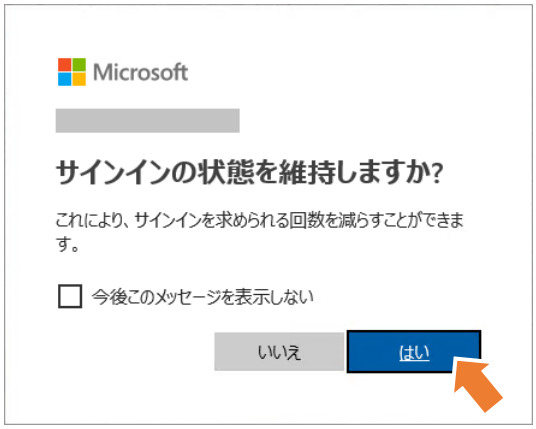
3. [サインイン] が表示されます。Microsoft アカウントのメール アドレスを入力し、[次へ] ボタンをクリックします。



4. パスワードを入力して、[サインイン] をクリックします。

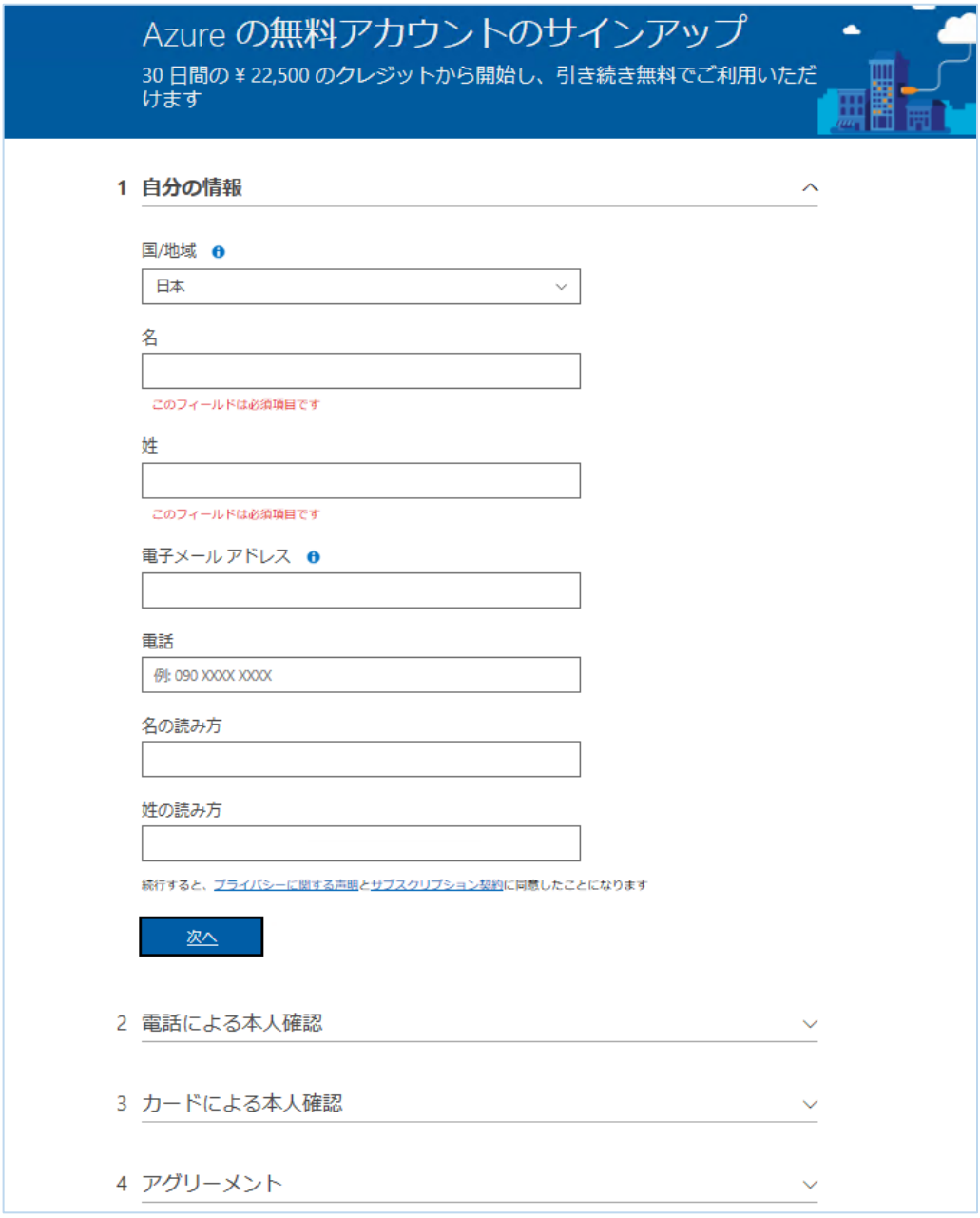


5. 「サインインの状態を維持しますか?」が表示されたら、[はい] をクリックします。



6. 使用している Microsoft アカウントに Azure サブスクリプションが紐付いていない場合、[Azure の無料アカウントのサインアップ] が表示されるので、サインアップに必要な以下の情報を設定します。

- 電話による本人確認
- カードによる本人確認
- アグリーメント



本人確認が完了したら、[アグリーメント] で [サインアップ] をクリックします。

2. Azure ポータルからのデータベースの作成と操作

ワン ポイント

Microsoft Azure の 12 か月間の無料評価版では、30 日間の ¥22,500 のクレジットと 12 が月間、記載している時間、および、容量まで以下サービスを使用ができます。

カテゴリ	Azure サービス名	使用可能な時間、サイズ
コンピューティング	Linux Virtual Machines	750 時間 (B1S)
	Windows Virtual Machines	750 時間 (B1S)
ストレージ	Managed Disks	P6 SSD 64 GB × 2
	Blob Storage	LRS ホット ブロック 5 GB
	File Storage	LRS 5 GB
データベース	SQL Database	250 GB
	Azure Cosmos DB	5 GB、400 RU/s
ネットワーク	Bandwidth (データ転送)	15 GB までの送信

Azure の無効アカウントを取得することで、Azure Cosmos DB は、400 RU/s の制限がありますが 5 GB のサイズまで使用できますので、有向活用してください。なお、無料評価版の利用は 1 回までとなっており、過去すでに利用された方は無料評価版にサインアップいただけませんので、ご注意ください。なお、無料評価版の使用制限に達した場合の動作については、以下の情報も参考にしてください。

- [お知らせ] 無料評価版の使用制限に達した場合の動作について

http://blogs.msdn.com/b/dsazurejp/archive/2012/12/28/windows-azure-90-days-free-offering.aspx

引き続き、ご使用される場合は、有償のサブスクリプションに切り替えていただきますようお願いいたします。また、既に MSDN Subscription をお持ちの方やスタートアップ企業の方、大学などの教育機関でご利用の方は、以下より詳細をご確認ください。

- MSDN サブスクライバー向けの Azure の特典

https://azure.microsoft.com/ja-jp/pricing/member-offers/msdn-benefits-details/

- スタートアップのための Azure

https://azure.microsoft.com/ja-jp/overview/startups/

- 教育機関でのご利用

https://www.microsoftazurepass.com/azureu

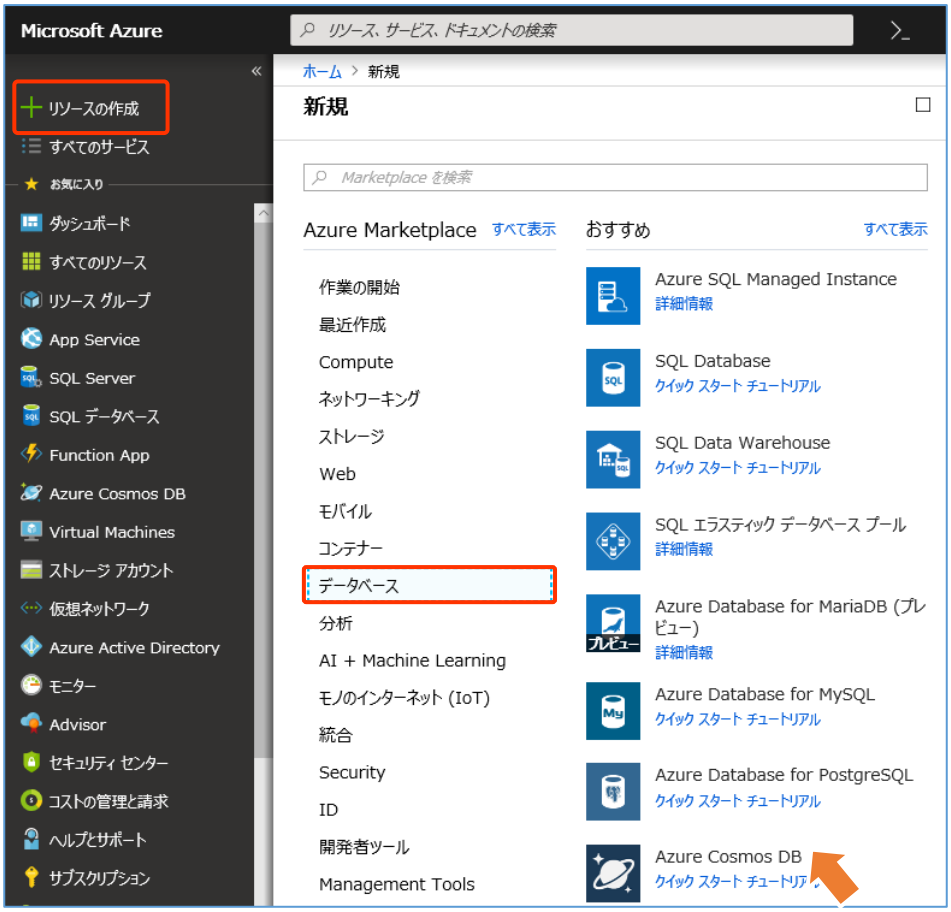
2.2 Azure ポータルからのデータベースの作成と検索

ここでは、グラフ データベースの操作に慣れていただくため、Azure ポータルから以下の操作を行います。

- Gremlin API を使用する Azure Cosmos DB データベース アカウントの作成
- グラフ データベースとコンテナーの作成
- Vertex の追加
- Edge の追加
- グラフ データの検索

2.2.1 Azure ポータルからの Azure Cosmos DB データベース アカウントの作成

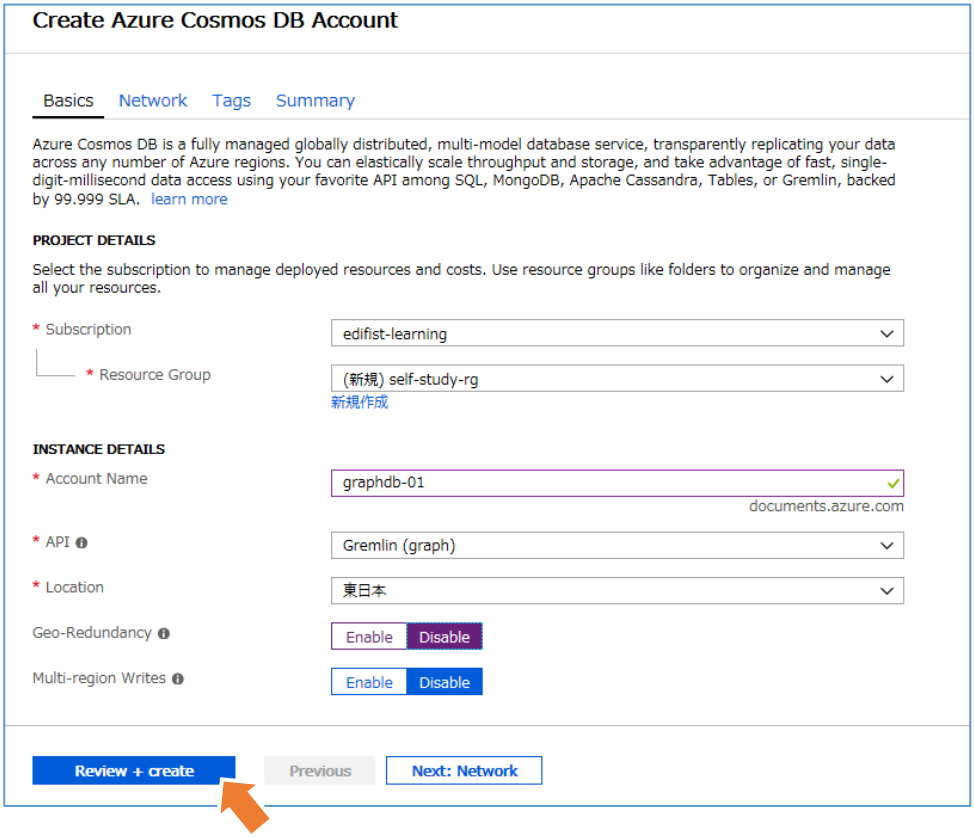
1. Azure ポータルのハブ メニューから [リソースの作成] ⇒ [データベース] ⇒ [Azure Cosmos DB] をクリックします。



2. Azure ポータルからのデータベースの作成と操作

2. [Create Azure Cosmos DB Account] ブレードの [Basics] ページが開くので、下表のパラメーター値を設定して [Review + create] ボタンをクリックします。

設定項目	設定値
Subscription	お持ちのサブスクリプション名
Resource Group	[新規作成] を選択して、自習書の作業手順で使用するためのリソースグループを作成してください
Account Name	Azure データ センター内で一意となる任意の値 (小文字・数値・ハイフン '-' のみを使用)
API	Gremlin (graph)
Location	選択可能な任意のリージョン
Geo-Redundancy	Disable
Multi-region Writes	Disable



ワン ポイント

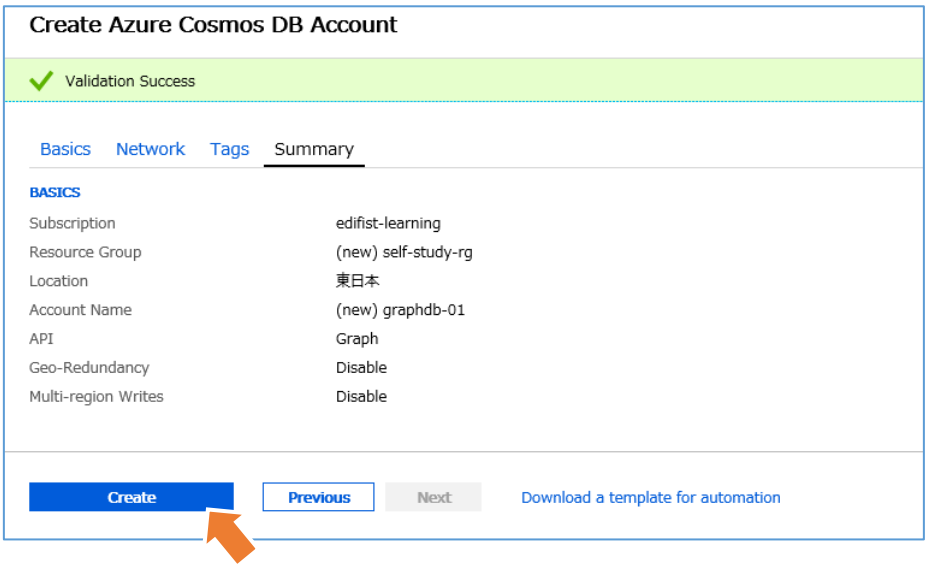
本自習書では、Azure Cosmos DB のアカウント名として、「graphdb-01」を使用していますが、アカウント名は Azure データ センター内で一意となる任意の値である必要があるため、既にこの名前が使用されている場合、異なる名前を使用していただく必要があることに注意してください。

なお「Geo-Redundancy」を有効化すると、プライマリ リージョンが存在する地域のペアとして、使用されるデータセンターにデータベースが複製されグローバル分散構成となります。可用性の SLA が、99.99 % から 99.999 % に向上しますが、料金は、2 倍のかかることに注意してください。

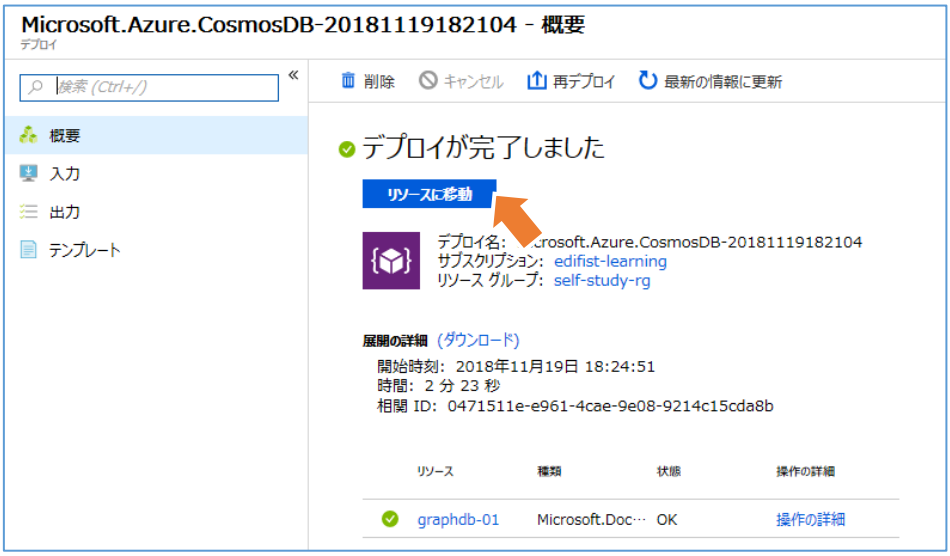
仮想ネットワークに配置したコンピューティング リソースからのアクセスさせる場合は、[Network] タブを使用して許可する Azure 仮想ネットワークを指定できます。

2. Azure ポータルからのデータベースの作成と操作

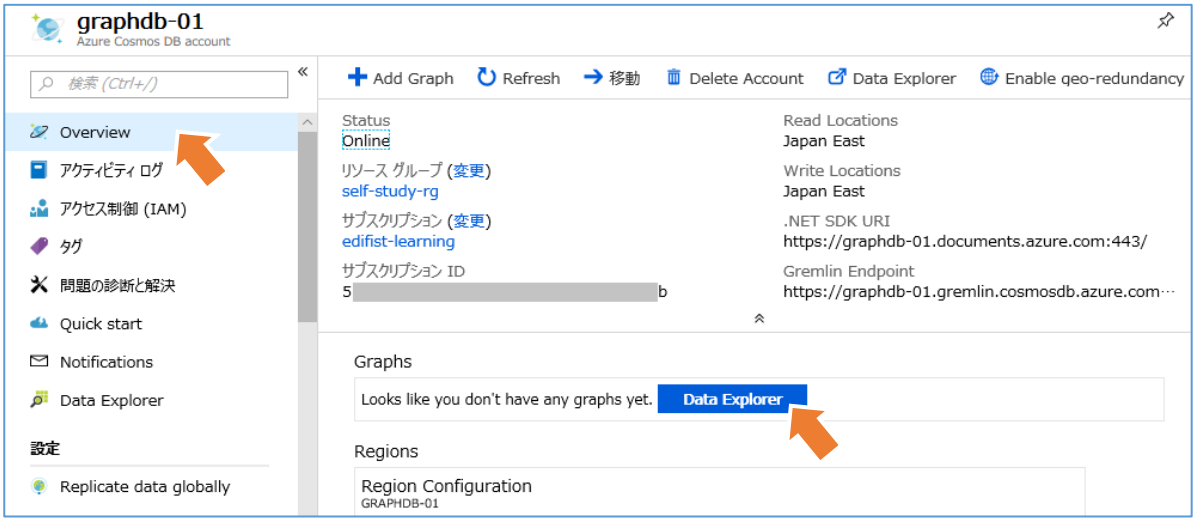
3. [Create Azure Cosmos DB Account] ブレードの [Summary] タブが表示されるので、設定容を確認して [Create] ボタンをクリックします。



4. 「デプロイが完了しました」が表示されたら、[リソースに移動] をクリックして、作成したリソースを開きます。

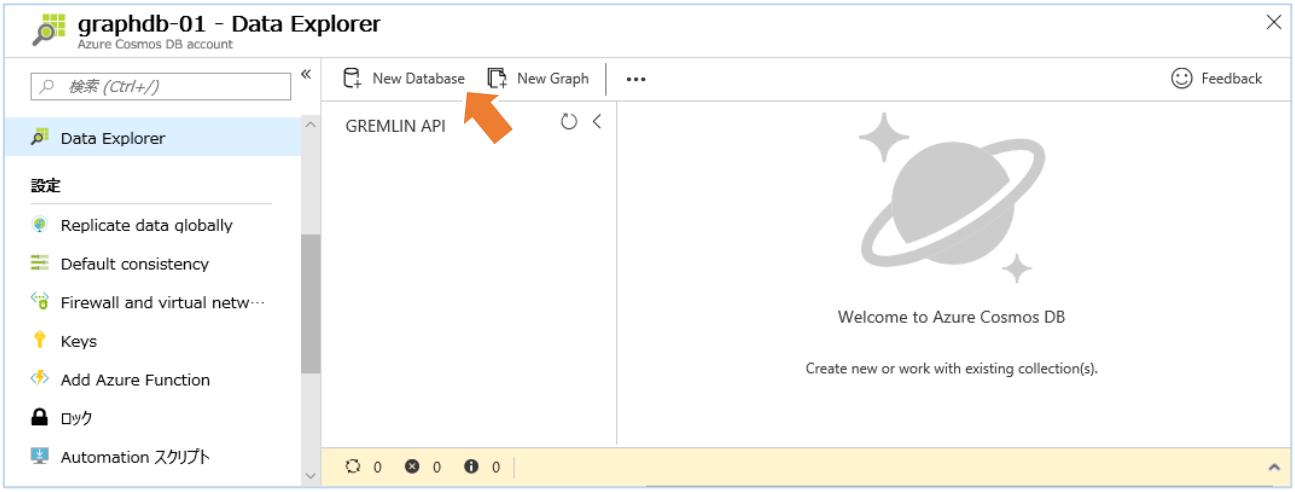


5. 作成した Azure Cosmos DB データベース アカウントの [Overview] ブレードを開き [Data Explorer] をクリックします。

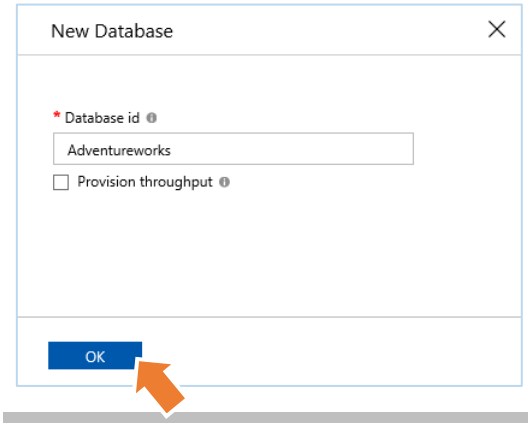


2.2.2 Azure ポータルからのグラフ データベースとコンテナーの作成

1. [Data Explorer] ブレードが表示されたら、新しいデータベースを作成するために、[New Database] をクリックします。



2. [New Database] が表示されるので [Database id] に「**Adventureworks**」と入力して [OK] ボタンをクリックします。



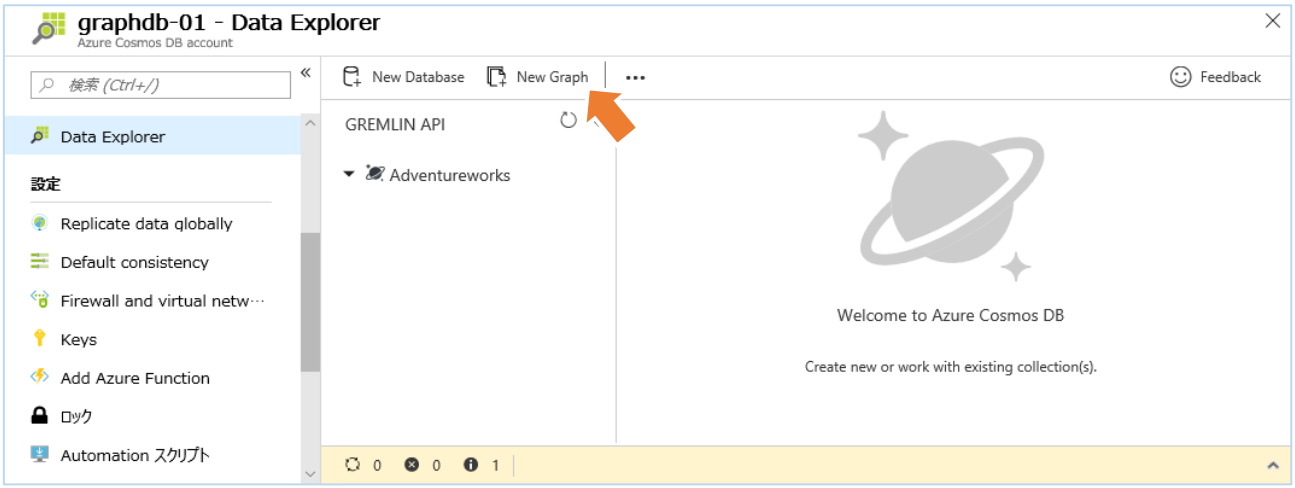
ワン ポイント

スループット (Throughput) は、1 秒あたりに処理する要求ユニットの数です。Azure Cosmos DB では、アプリケーションが利用する要求ユニットの量を秒単位で予約できます。Azure Cosmos DB での各操作は、CPU、メモリ、IOPS が消費されます。各操作により、要求されるリソースの使用量を要求ユニットとして表現しています。要求ユニット使用量に影響を及ぼす要因とアプリケーションのスループット要件を理解しておくと、アプリケーション実行のコスト効率を向上できます。

[Provision thoughput] を選択すると、データベース レベルで、スループットを予約し、データベースに作成されるコレクション (グラフ) で共有することができます。

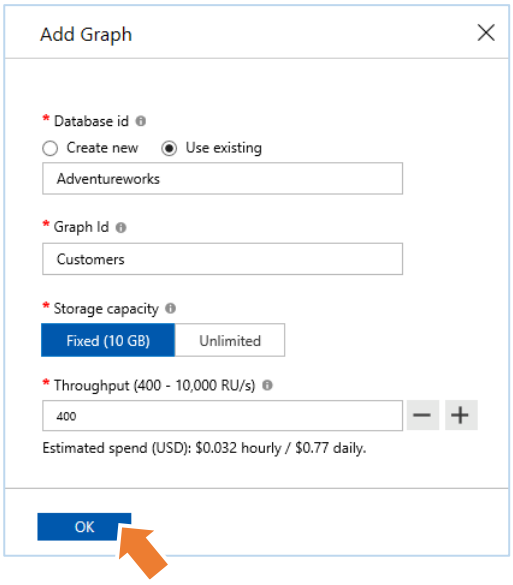
2. Azure ポータルからのデータベースの作成と操作

3. Adventureworks という名前で、データベースが作成されたことを確認して、[New Graph] をクリックします。



4. [Add Graph] が表示されるので、下表のパラメーター値を設定して [OK] をクリックします。

設定項目	設定値
Database id	[Use existing] オプションを選択して、「Adventureworks」を選択します。
Graph Id	「Customers」と入力します。
Storage capacity	[Fixed (10 GB)] を選択します。
Throughput	「400」と入力します。



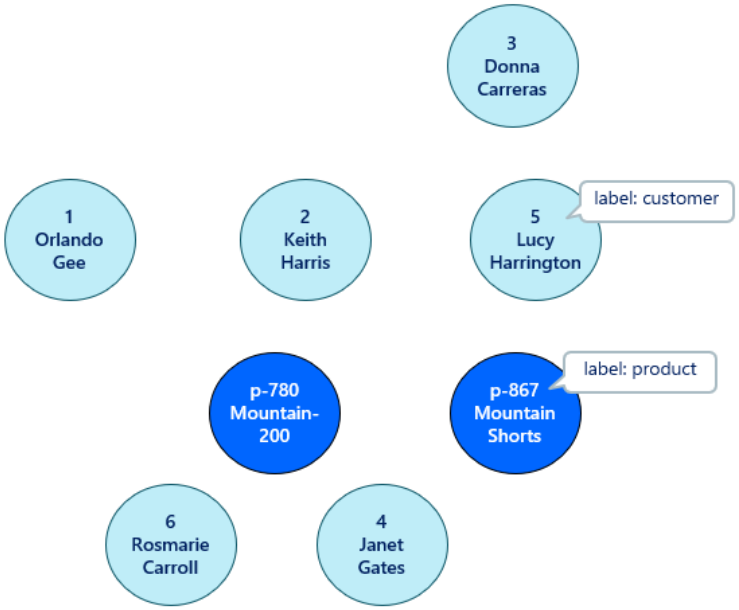
ワン ポイント

[Storage capacity] で [Fixed (10 GB)] を選択すると最大で 10 GB までのグラフを格納できるコンテナーが作成されます。スループット (Throughput) は、1 秒あたり最大で 10,000 の要求単位まで割り当てられます。[Storage capacity] で [Unlimited] を選択すると、行方向のパーティション分割が可能になり 10 GB の制限を超えて、グラフを格納できるように自動的にスケーリングが行われます。[Unlimited] を選択する場合は、Vertex の id および label 以外のプロパティをパーティション キーとして指定してください。グラフ データのパーティション分割については、Azure Cosmos DB でのパーティション分割されたグラフの使用を参照してください。

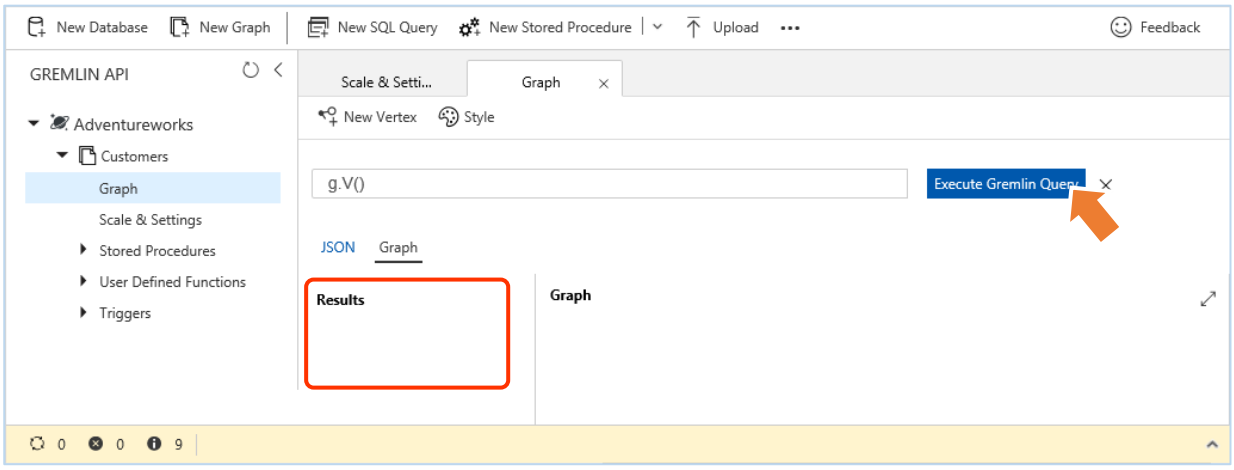
固定で作成したコンテナーを後から、無制限に変更することができないため、コンテナー作成の前に、そちらの種類で実装するのかを決定してください。

2.2.3 Data Explorer による Vertex の追加

ここから、Azure ポータルの Data Explorer を使用して、グラフ操作を行います。最初にラベルを customer に設定した Vertex をグラフに追加します。その後に、ラベルを product に設定した Vertex をグラフに追加します。

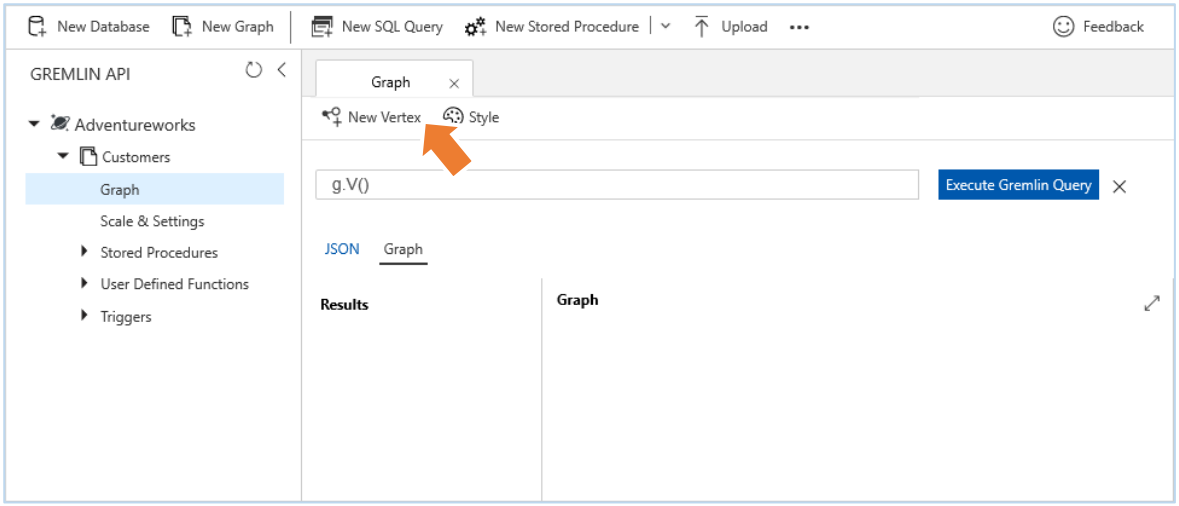


1. [Data Explorer] ブレードに表示されるツリーで Graph ノードをクリックした後、既定のグラフ クエリとして、「**g.V()**」がセットされていることを確認して、[Execute Gremlin Query] をクリックして実行します。



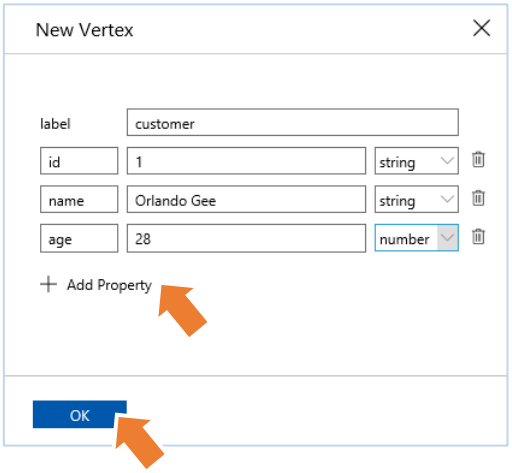
ワン ポイント
Data Explorer に既定で設定されている「g.V()」を実行すると、グラフからすべての Vertex を取得できます。ここでは、まだ、Vertex を追加していないため、[Results] 領域には何も表示されません。

2. Data Explorer から customer ラベルの Vertex を追加するため、[New Vertex] をクリックします。



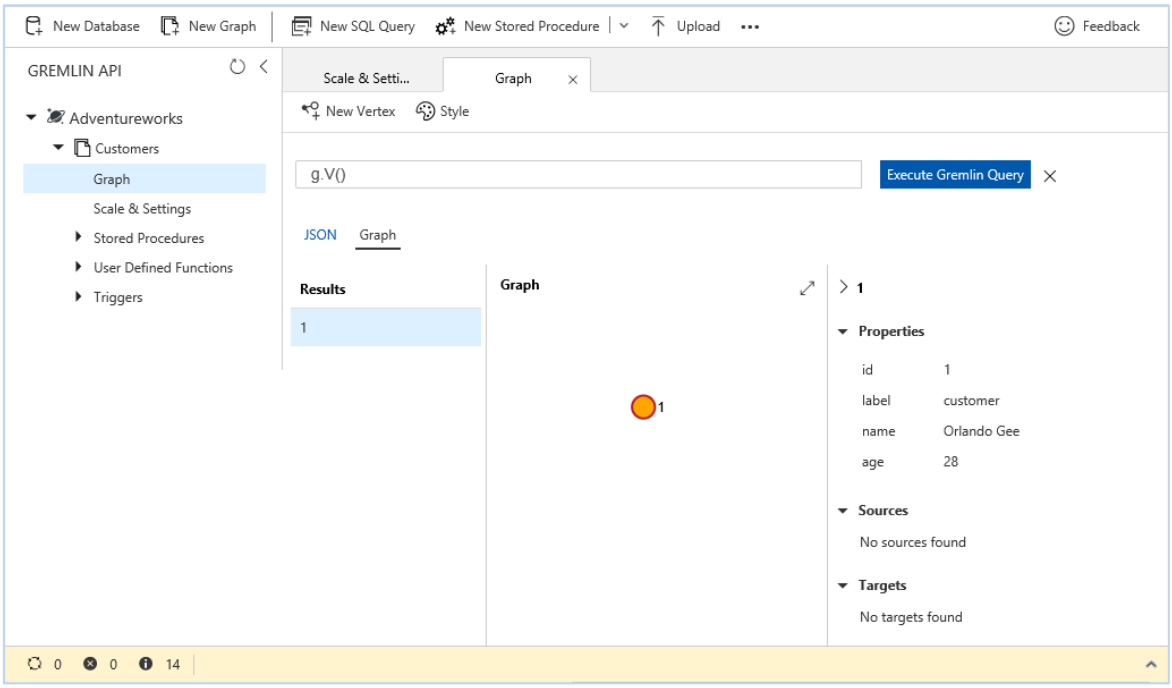
3. [New Vertex] で、[label] に「**customer**」と入力して [Add Property] を 3 回クリックした後、Vertex のプロパティとして使用する 3 種類のキー/バリュー値を追加して、[OK] ボタンをクリックします。

Key	Value	Type
id	1	string
name	Orlando Gee	string
age	28	number



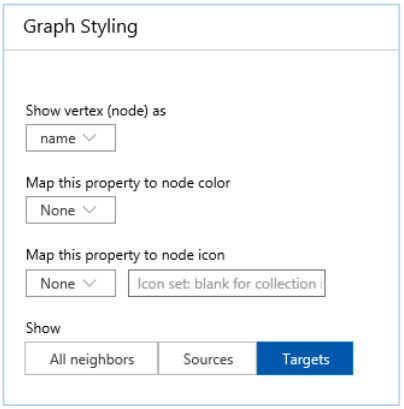
ワン ポイント
ここで、id の設定を省略することができますが、その場合、内部的に生成された GUID 値が、id として使用されることに注意してください。

4. customers グラフに 1 件の顧客データが格納されたことを確認します。



ワン ポイント

Graph タブには、作成された Vertex の情報が表示されます。右側に id、label、name、age といったプロパティが表示されます。また、[Style] を使用して、表示される情報を変更することができます。



次に、Data Explorer からグラフ クエリを実行して Vertex を追加します。Vertex を追加するグラフ クエリの構文は、以下になります。g はグラフを意味し、addV() はクエリ言語で「ステップ」と呼ばれています。addV() は、Vertex を追加するステップです。

```
g.addV(<ラベル>).property(<キー>,<バリュー>).property(<キー>,<バリュー>).property(<キー>,<バリュー>)
```

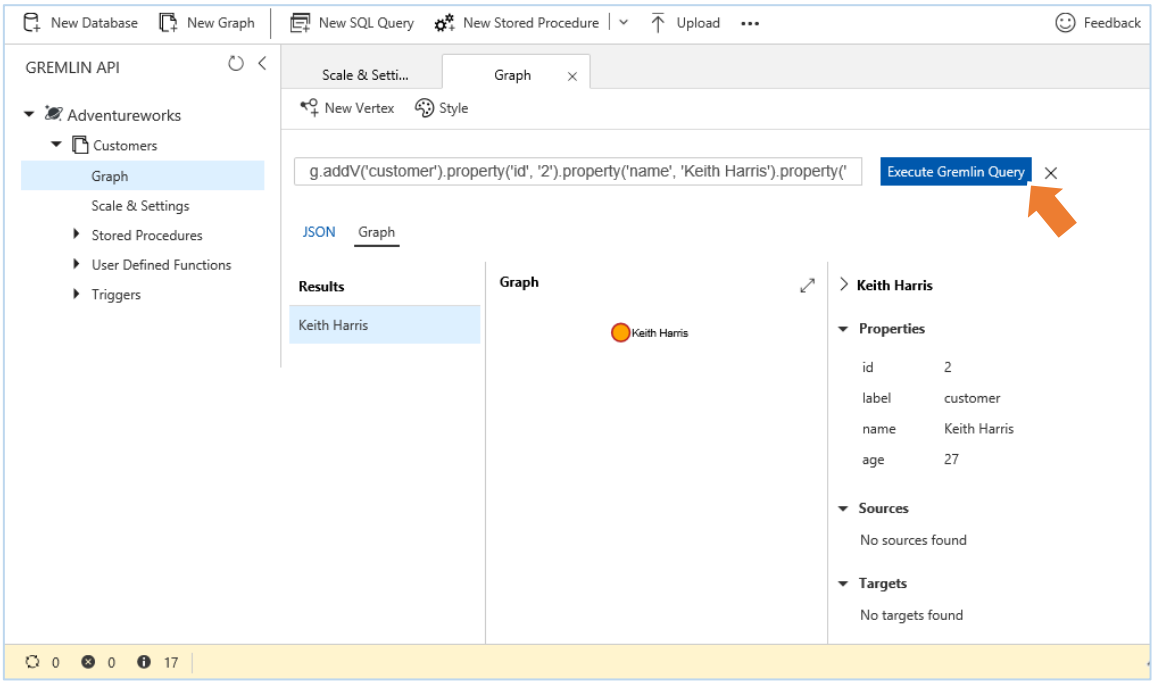
ワン ポイント

手順 3 で行った操作をグラフ クエリで記述すると、

「g.addV('customer').property('id', '1').property('name', 'Orlando Gee').property('age', 28)」となります。

5. 2 個目の Vertex の追加をグラフ クエリを使用して行うために、既定のグラフ クエリの「**g.V()**」を以下に書き換えた後、[Execute Gremlin Query] をクリックして実行します。

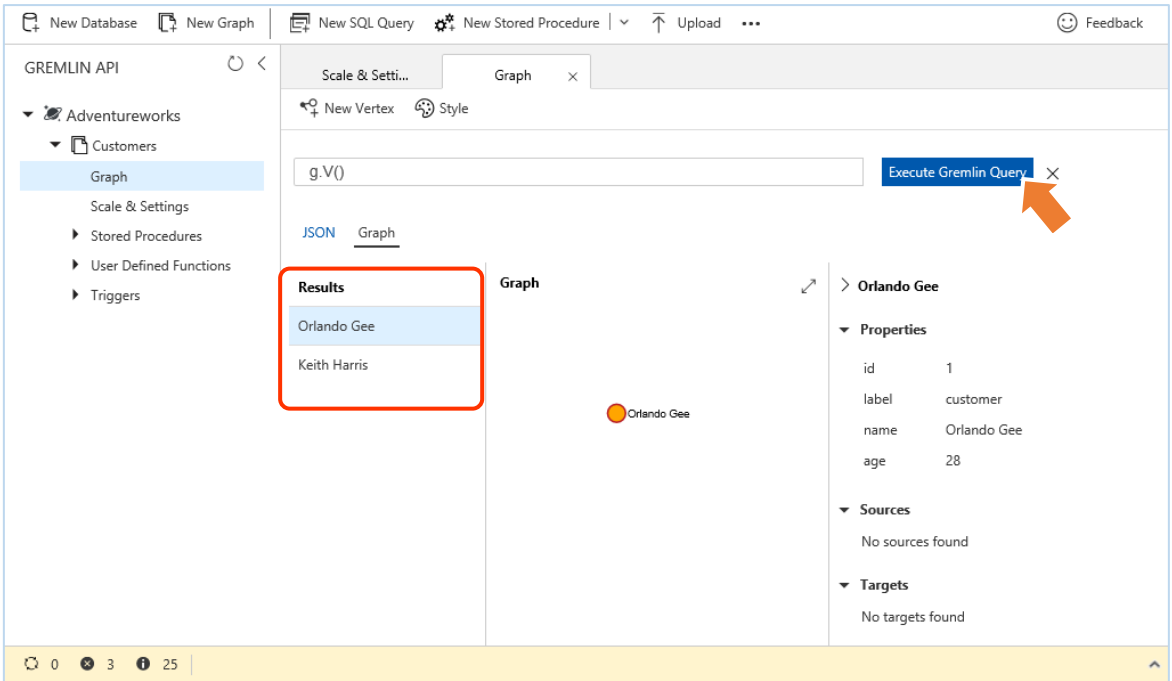
```
g.addV('customer').property('id', '2').property('name', 'Keith Harris').property('age', 27)
```



ワン ポイント

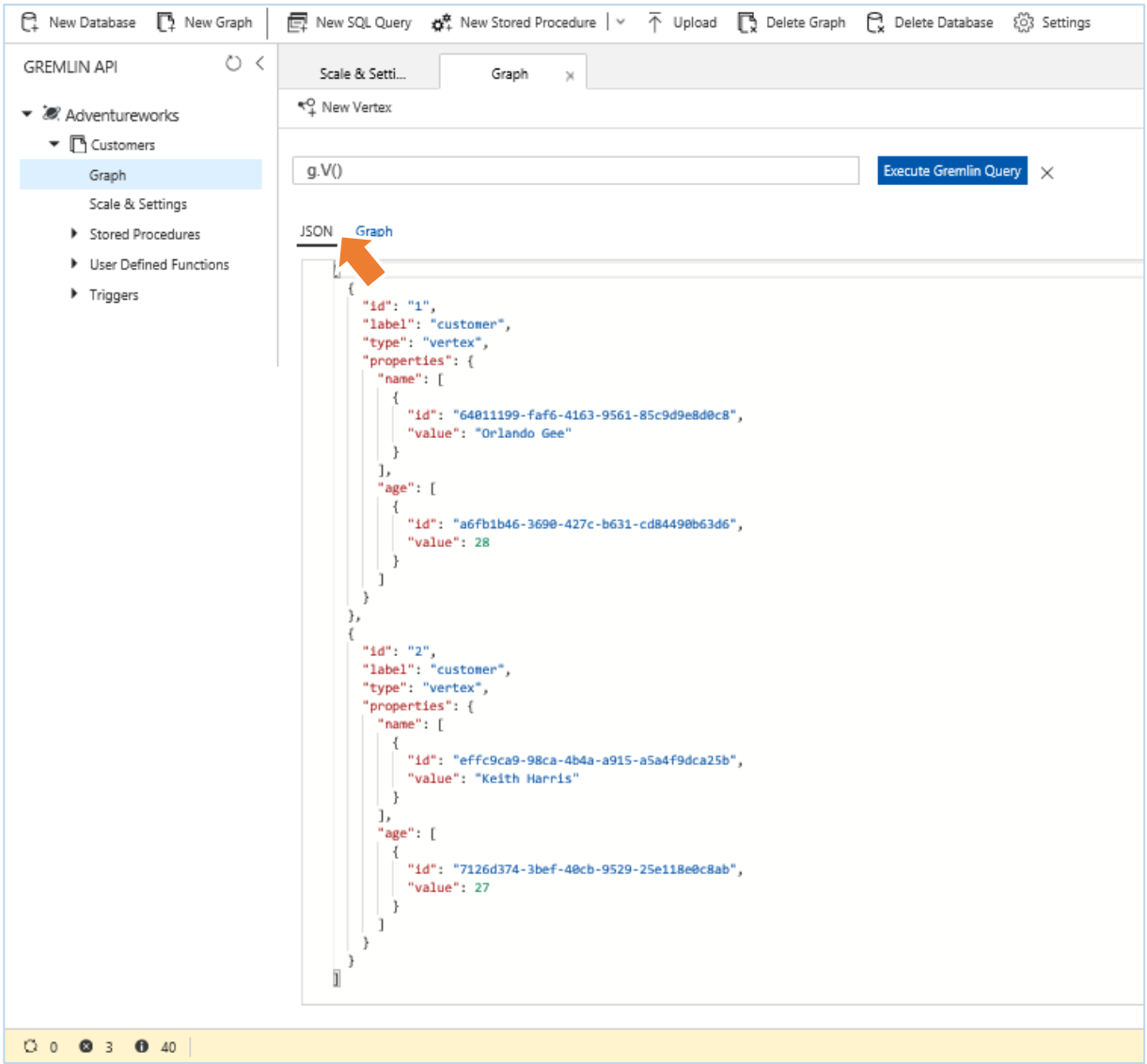
addV() ステップを使用すると引数で指定されたラベルを持つ Vertex を追加できます。Vertex のプロパティは、キー/バリュー形式で property() に記述します。なお、作成した Vertex を削除するには、対象の Vertex の id を指定して「g.V(<Vertex の id >).drop()」を実行します。

6. ここでグラフからすべての Vertex を取得するクエリを再度、実行するため、グラフ クエリを「**g.V()**」に書き換えた後、[Execute Gremlin Query] をクリックします。



ワン ポイント
ここまでの作業でグラフに追加された 2 個の customer ラベルの Vertex の name プロパティが [Results] 領域に表示されます。Vertex のフォーカスを Orlando Gee から Keith Harris に切り替えると、連動して右側に表示されるプロパティ表示が切り替わります。

7. ここで [JSON] タブを選択します。



ワン ポイント
Azure Cosmos DB は、グラフ データベースから操作結果を返すときに GraphSON 形式を使用します。GraphSON は、JSON を使用して Vertex、Edge、プロパティ (単一値および複数値プロパティ) を表すためのグラフ データベースの標準形式です。Azure Cosmos DB では、Data Explorer の [Upload] ボタンを使用することで、GraphSON ドキュメントをアップロードして、Vertex や Edge を追加することもできます。また、[New SQL Query] ボタンを使用するとグラフが内部的に保持している JSON ドキュメントに対し、SQL クエリを実行することもできます。 なお、現時点では、Azure Cosmos DB Gremlin API グラフ データベースへのデータ移行については、「データ移行ツールを使用して Azure Cosmos DB にデータを移行する」ページから入手できる移行ツール (Dtui.exe および Dt.exe) が対応していないことに注意してください。 大量のデータをグラフ データベースに移行する場合、BulkExecutor .NET ライブラリを利用したコードを記述する必要が

あります。BulkExecutor .NET ライブラリについては、「グラフの BulkExecutor .NET ライブラリを使って Azure Cosmos DB Gremlin API の一括操作を実行する」を参照してください。

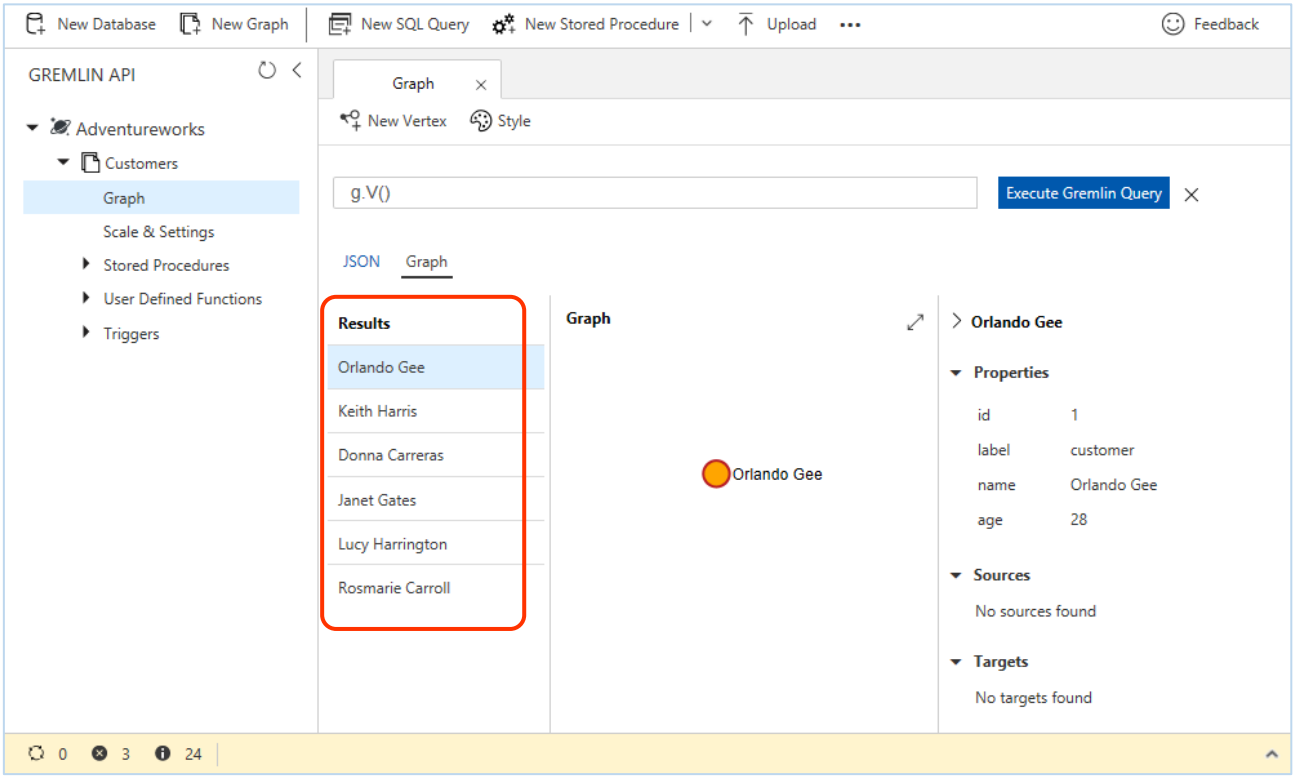
8. さらに 4 個の customer ラベルの Vertex を追加するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックしながら、順次実行します。

```
g.addV('customer').property('id', '3').property('name', 'Donna Carreras').property('age', 32)
g.addV('customer').property('id', '4').property('name', 'Janet Gates').property('age', 37)
g.addV('customer').property('id', '5').property('name', 'Lucy Harrington').property('age', 32)
g.addV('customer').property('id', '6').property('name', 'Rosmarie Carroll').property('age', 27)
```

ワン ポイント

まとめて、実行するとエラーになるため 1 行ずつ実行してください。

9. ここまでの結果を確認するため、[Data Explorer] ブレードで、「**g.V()**」と入力して、[Execute Gremlin Query] をクリックしてグラフ クエリを実行します。



ワン ポイント

グラフに追加された 6 個の customer ラベルの Vertex の name プロパティが [Results] 領域に表示されます。

10. 次に 2 個の product ラベルの Vertex を追加するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックしながら、順次実行します。

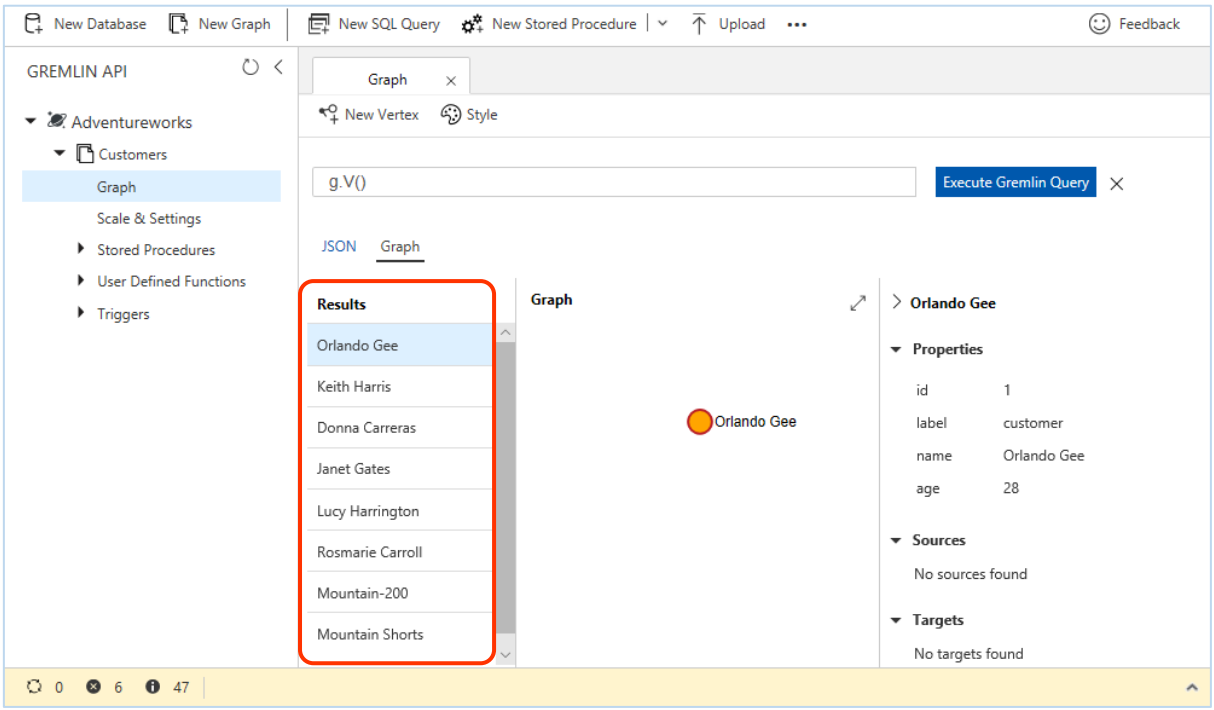
```
g.addV('product').property('id', 'p-780').property('name', 'Mountain-200').property('price', 2319.99)
g.addV('product').property('id', 'p-867').property('name', 'Mountain Shorts').property('price', 69.99)
```

ワン ポイント

2. Azure ポータルからのデータベースの作成と操作

id 値は、ラベルが異なったとしても、コンテナー内 (パーティション分割している場合は、パーティション内) で一意でなければならないことに注意してください。自習書で使用しているデータは、Adventureworks サンプルデータベースのテーブルのデータを流用していますが、customer ラベルの Vertex の id 値と重複を避けるため、product ラベルの Vertex では、id 値に接頭辞として「p-」を付加しています。

11. ここまでの結果を確認するために、[Data Explorer] ブレードで、「**g.V()**」と入力して、[Execute Gremlin Query] をクリックしてグラフ クエリを実行します。

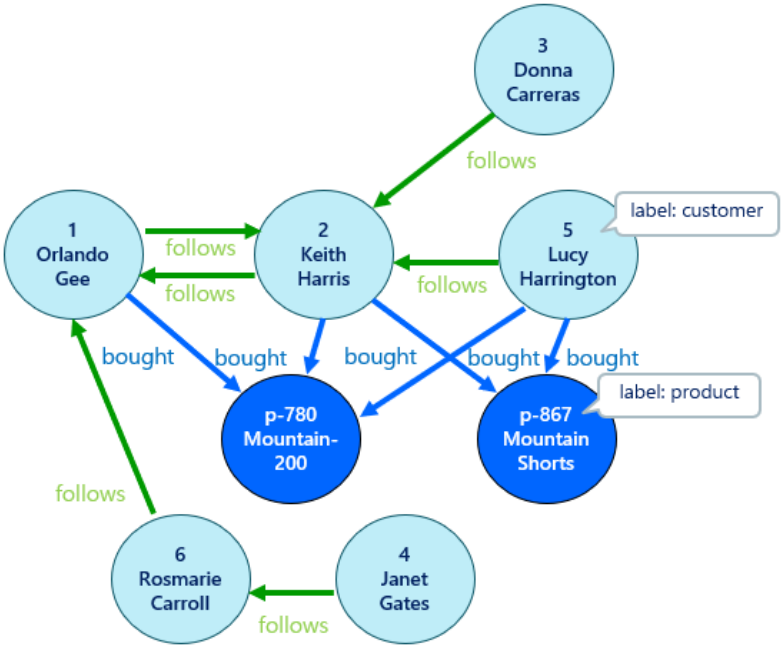


ワン ポイント

グラフに追加された 6 個の customer ラベルの Vertex と 2 個の product ラベルの Vertex の name プロパティが [Results] 領域に表示されます。

2.2.4 Data Explorer による Edge の追加

ここまでの作業で必要な Vertex がすべて作成されましたので、ここからは、下図の構成で Edge を追加します。



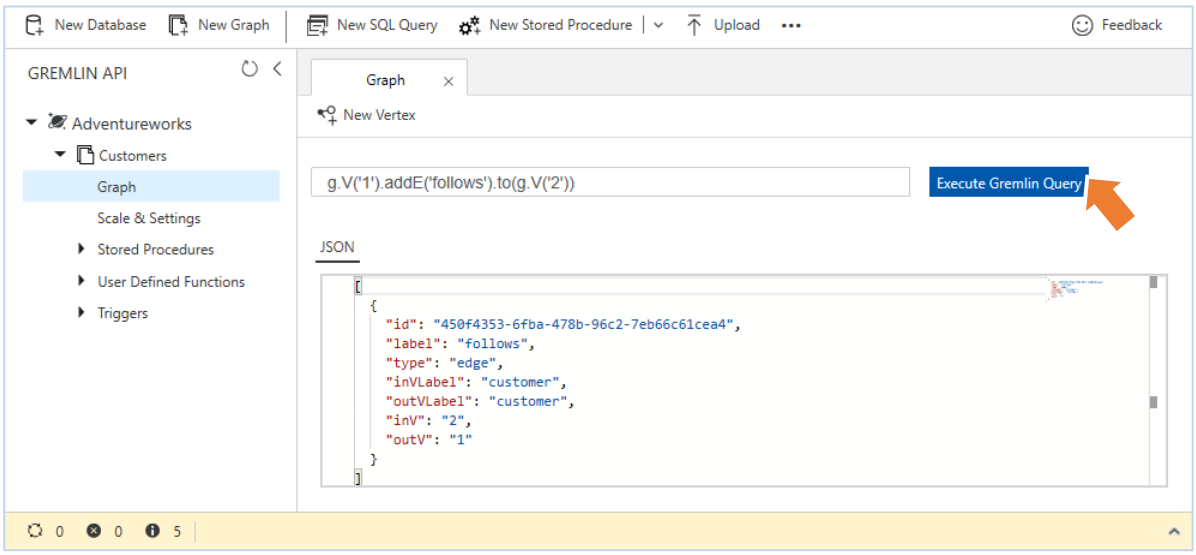
2018 年 12 月の時点では、Data Explorer には、Edge を追加するための UI が提供されていないため、グラフ クエリを使用します。Edge を追加するクエリの構文は以下になります。

```
g.V(<開始点の Vertex の id 値>).addE(<ラベル>).to(g.V(<到達点の Vertex の id 値>)).property(<キー>,<バリュー>)
```

Edge は、2 つの Vertex の関係性を表現します。関係性には方向を伴います。ですから、Edge の追加では、開始点となる Vertex に対して、addE() ステップで、Edge のラベルを記述し、to() には、矢印が向けられる到達点の Vertex を記述します。Edge のプロパティ値は、Vertex の時と同じように、キー/バリュー形式で property() に記述します。

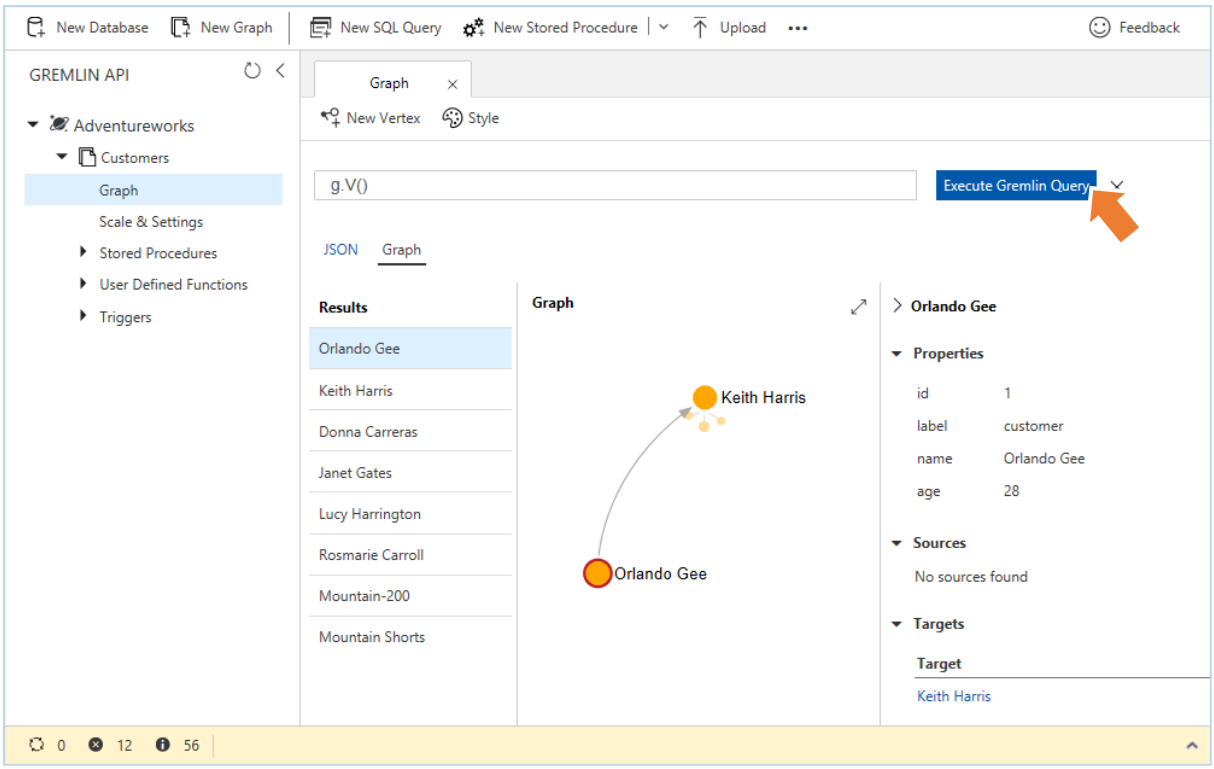
1. まず、Orlando Gee から Keith Harris への follows ラベルの Edge を追加するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V('1').addE('follows').to(g.V('2'))
```



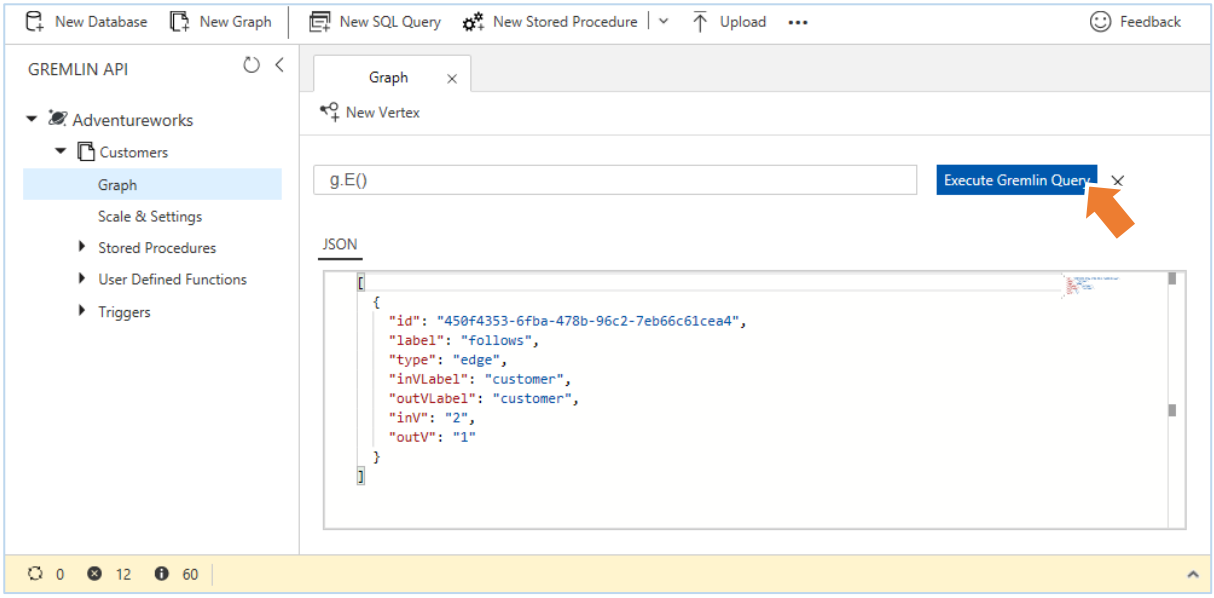
ワン ポイント
Edge を追加するグラフ クエリを実行すると、上記のような GraphSON が返ります。Vertix オブジェクトが返されるグラフ クエリの実行では、グラフが出力されていましたが、Vertix オブジェクトのプロパティや Edge の情報を取得した場合、GraphSON 形式が返されます。 追加された Edge の、inVLabel と outVLabel が customer で inV が 2、outV が 1 のため、id が 1 の customer から 2 の customer への Edge であることがわかります。Edge の Id は、明示的に記述していないため、GUID 値が生成されています。Id を指定する場合、property() を使用し、「g.V('1').addE('follows').to(g.V('2')).property('id', '1001')」のように記述します。

2. グラフに作成した Vertex を確認するために、グラフ クエリとして「**g.V()**」と入力し、[Execute Gremlin Query] をクリックします。



ワン ポイント
「**g.V()**」は、グラフの Vertex を取得します。Edge で結ばれた Vertex は、その関係性が表示されます。V() ステップで取得される GraphSON に、Edge の定義は含まれていませんが、内部的には、Vertex と一緒に Edge の情報も GraphSON 形式で格納されます。Edge の情報を GraphSON で取得するには、E() ステップを使用し、グラフからすべての Edge を取得する場合、「**g.E()**」を実行します。

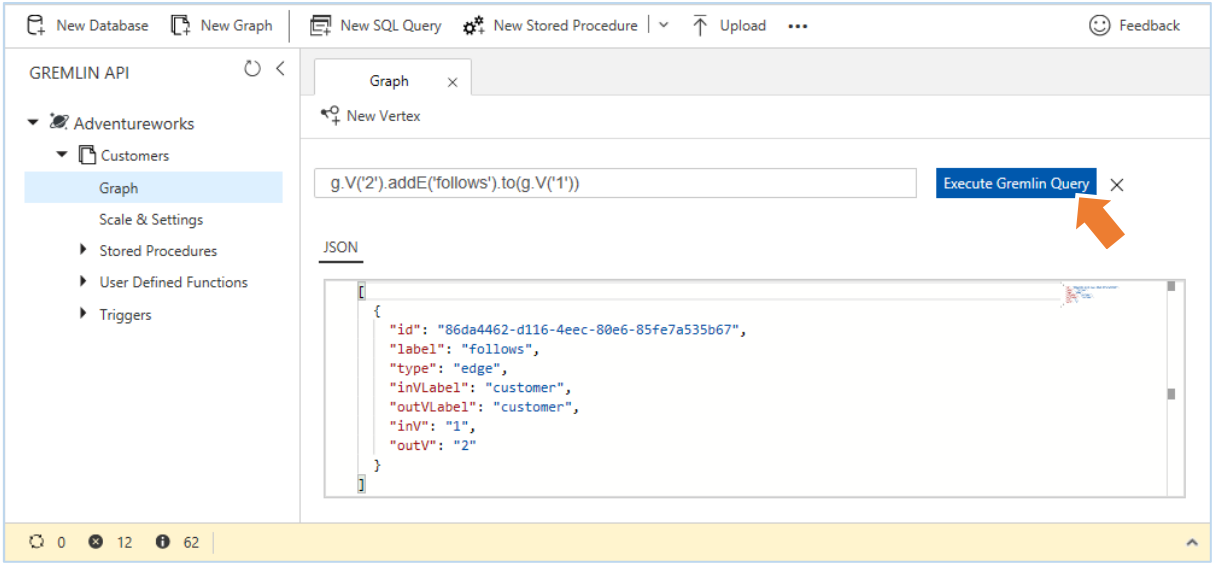
3. 追加した Edge を確認するために、グラフ クエリとして「**g.E()**」と入力して、[Execute Gremlin Query] をクリックします。



ワン ポイント
Edge を作成したときに返された GraphSON が表示されます。

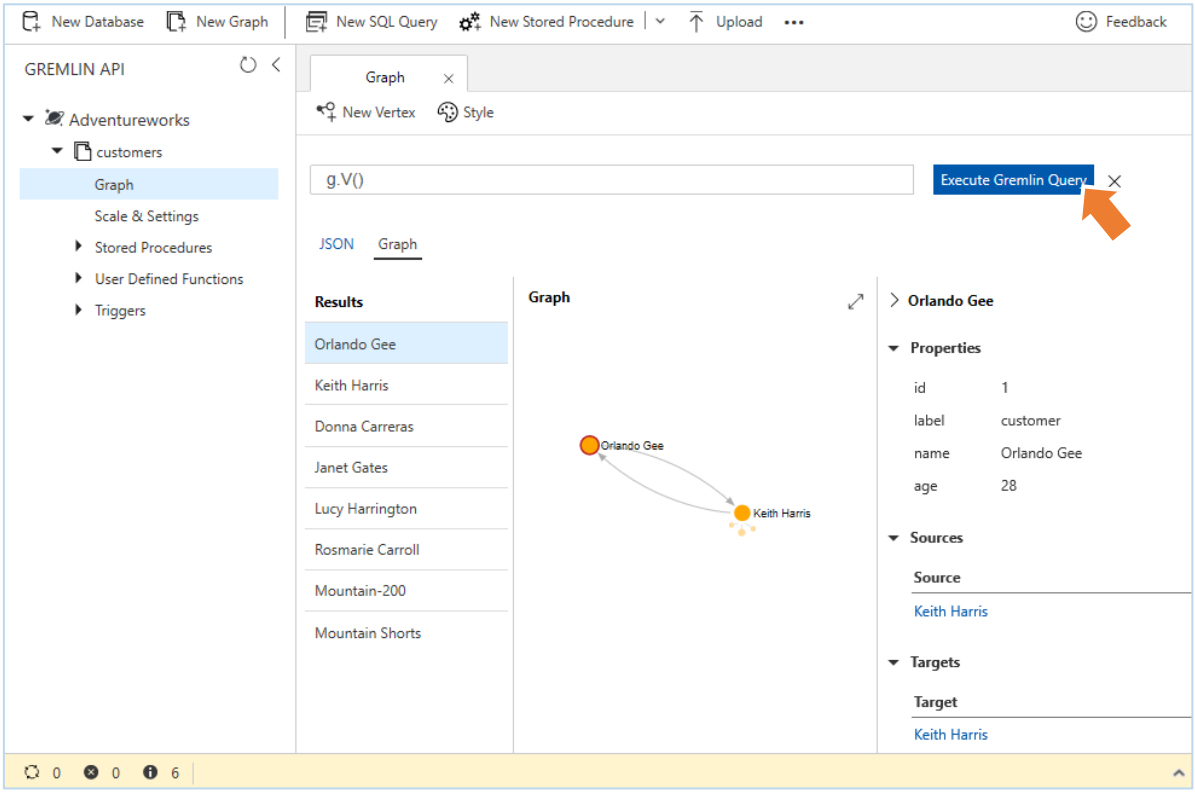
4. 次に、Keith Harris から Orlando Gee への follows ラベルの Edge を追加するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V('2').addE('follows').to(g.V('1'))
```



ワン ポイント
手順 1 で実行したクエリとは逆方向の Edge になります。

5. グラフに作成した Vertex を確認するために、グラフ クエリとして「**g.V()**」と入力して、[Execute Gremlin Query] をクリックします。

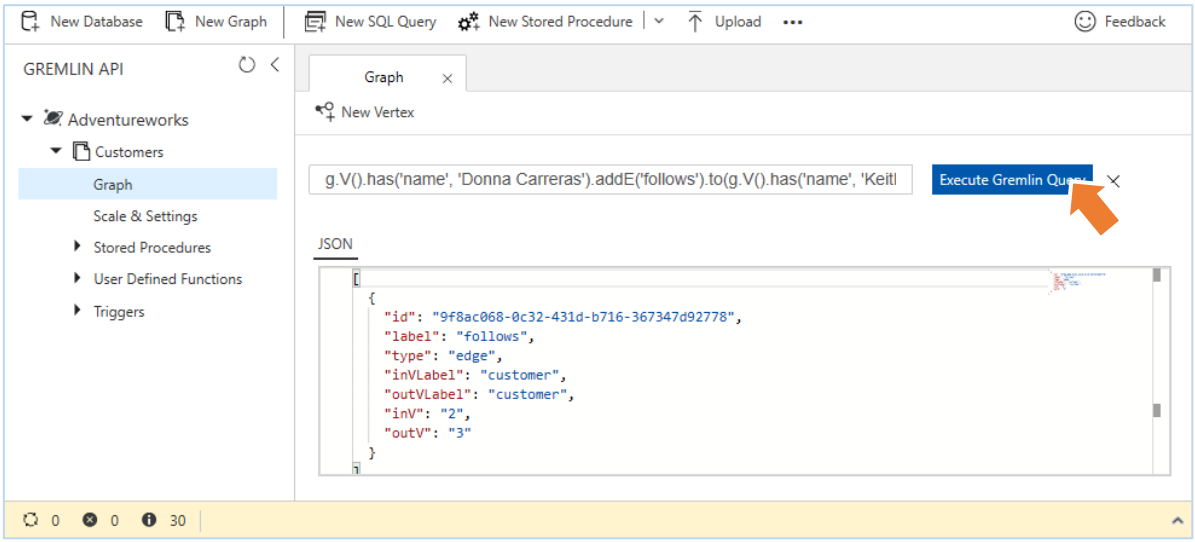


ワン ポイント
Orlando Gee と Keith Harris の間で双方向に Edge が追加されていることがわかります。Edge の追加では、対象となる Vertex に対し一意な値が保証される id を使用することが推奨されますが、他に一意であるプロパティがある場合、例えば、name プロパティが一意であれば、「g.V('2').addE('follows').to(g.V('1'))」と書く代わりに、「g.V().has('name', 'Keith Harris').addE('follows').to(g.V().has('name', 'Orlando Gee'))」と書くことができます。

2. Azure ポータルからのデータベースの作成と操作

6. id の代わりに name プロパティを使用して、follows ラベルの Edge を追加するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().has('name', 'Donna Carreras').addE('follows').to(g.V().has('name', 'Keith Harris'))
```



ワン ポイント

このクエリは、name プロパティの値が一意であれば、「g.V('3').addE('follows').to(g.V('2'))」と等しく動作します。また、name プロパティの値が一意ではない場合、以下のように Vertex をフィルターするプロパティを追加することもできます。「g.V().has('name', 'Donna Carreras').has('age', 32).addE('follows')
.to(g.V().has('name', 'Keith Harris').has('age', 27))」
has() ステップは、Vertex や Edge のプロパティをフィルター処理するときに使用できます。id やラベルを対象としてフィルター処理する場合は、hasId()、hasLabel() ステップを使用できます。

7. さらに 3 個の follows ラベルの Edge を追加するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックしながら、順次実行します。

```
g.V().has('name', 'Janet Gates').addE('follows').to(g.V().has('name', 'Rosmarie Carroll'))
g.V().has('name', 'Lucy Harrington').addE('follows').to(g.V().has('name', 'Keith Harris'))
g.V().has('name', 'Rosmarie Carroll').addE('follows').to(g.V().has('name', 'Orlando Gee'))
```

ワン ポイント

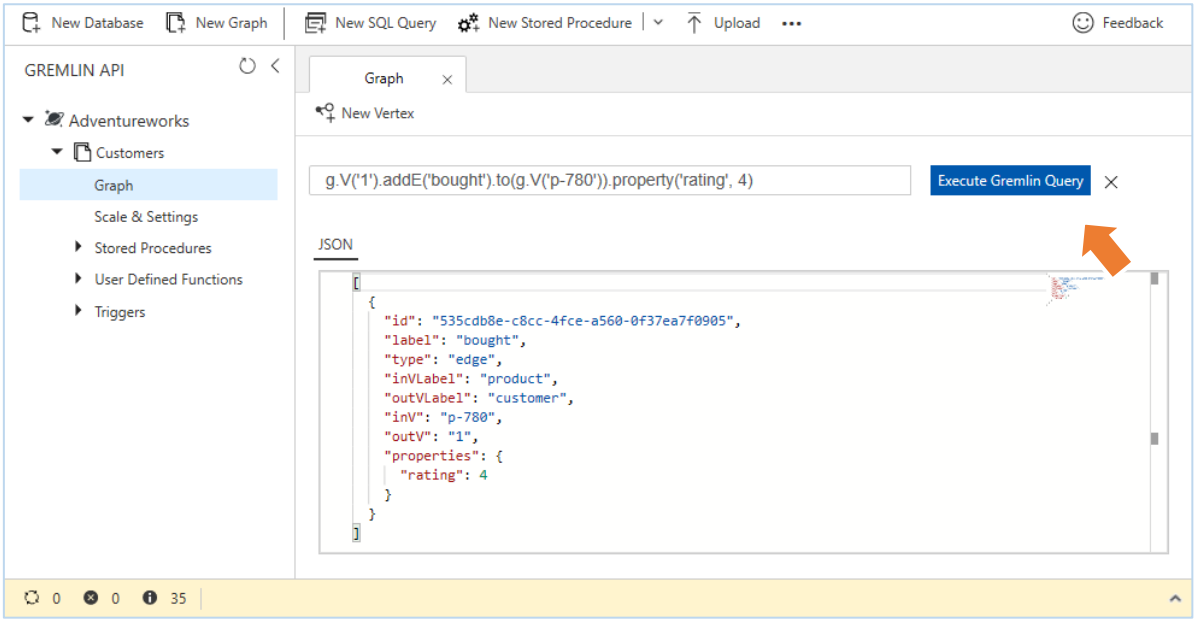
これらのクエリは、以下のクエリと同じ動作をします。
g.V('4').addE('follows').to(g.V('6'))
g.V('5').addE('follows').to(g.V('2'))
g.V('6').addE('follows').to(g.V('1'))

次に顧客が商品を購入したことを示す bought ラベルの Edge を追加します。顧客は購入した商品を 5 段階で評価していることとして評価の値を rating プロパティに保持するようにします。

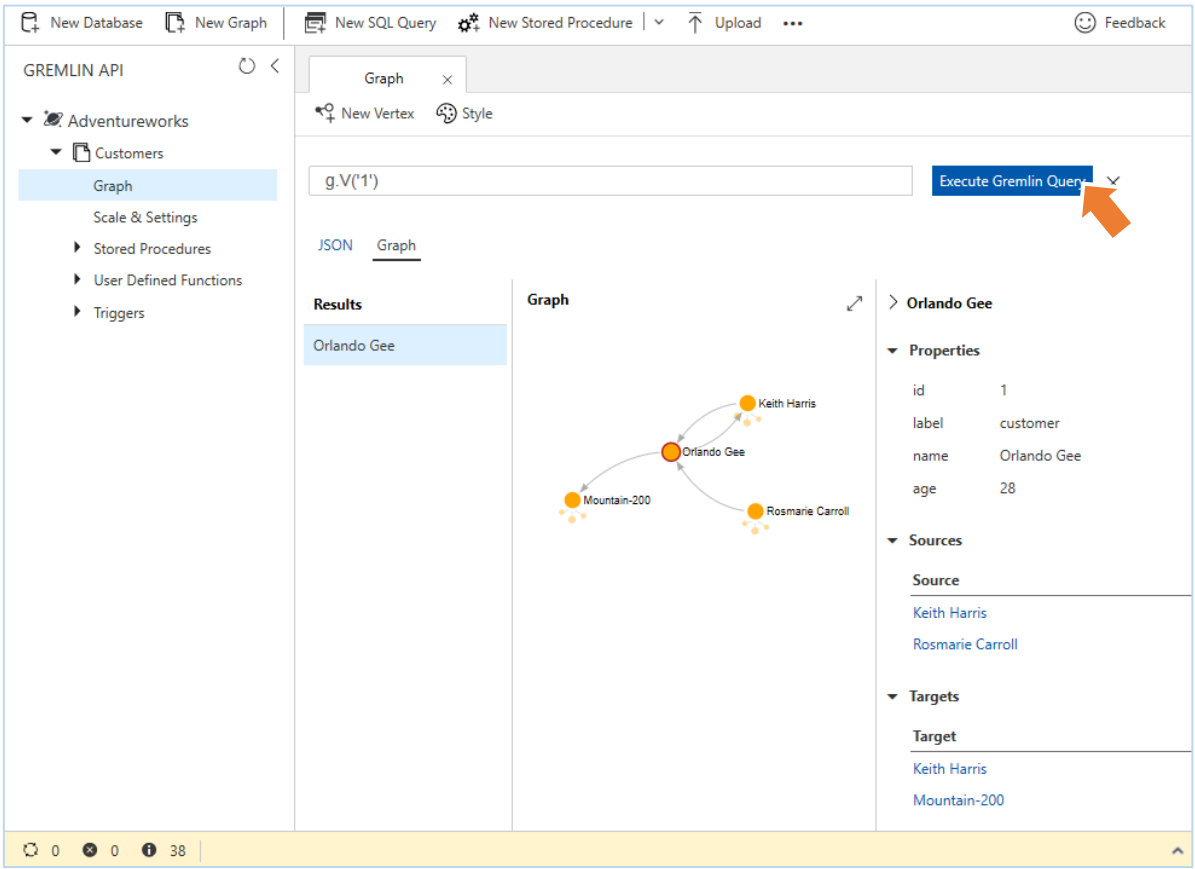
8. Orlando Gee (id=1) が、Mountain-200 (id=p-780) のマウンテン バイクを購入したことを示す bought ラベルの Edge を追加するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V('1').addE('bought').to(g.V('p-780')).property('rating', 4)
```

2. Azure ポータルからのデータベースの作成と操作



9. Orlando Gee (id=1) を中心としたエンティティのつながりを確認するために、グラフ クエリとして「**g.V('1')**」と入力して、[Execute Gremlin Query] をクリックします。



ワン ポイント
Orlando Gee (id=1) がフォローしている顧客と、フォローしている顧客に加えて、商品の購入を示す Edge が追加されていることがわかります。

10. さらに 4 個の bought ラベルの Edge を追加するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックしながら、順次実行します。

```
g.V('2').addE('bought').to(g.V('p-780')).property('rating', 5)
g.V('5').addE('bought').to(g.V('p-780')).property('rating', 4)
g.V('2').addE('bought').to(g.V('p-867')).property('rating', 3)
g.V('5').addE('bought').to(g.V('p-867')).property('rating', 5)
```

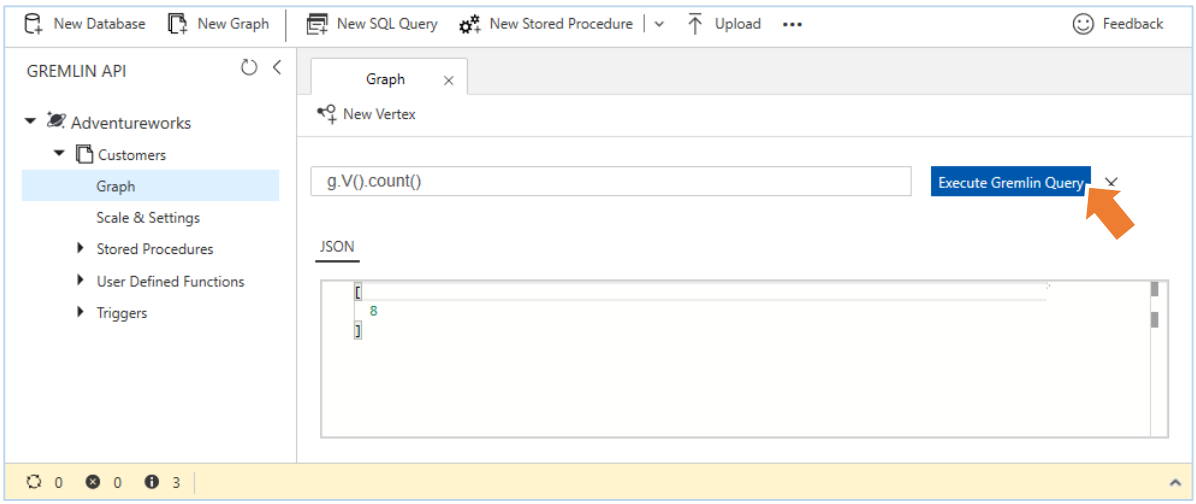
2.2.5 Data Explorer によるグラフのトラバーサル

ここでは、Azure ポータルの Data Explorer から、グラフ クエリを実行して、ここまでの作業で追加した Vertex と Edge を使用したクエリを実行しながら、主要なステップの役割を理解していきます。基本的なクエリから始めて、第 1 章で紹介したようなビジネス要件を満たすためのグラフ クエリまで順番に試して行きましょう。

2.2.5.1 基本的なグラフ クエリ

1. ここまでの手順で、グラフに追加した Vertex の数をカウントするために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().count()
```



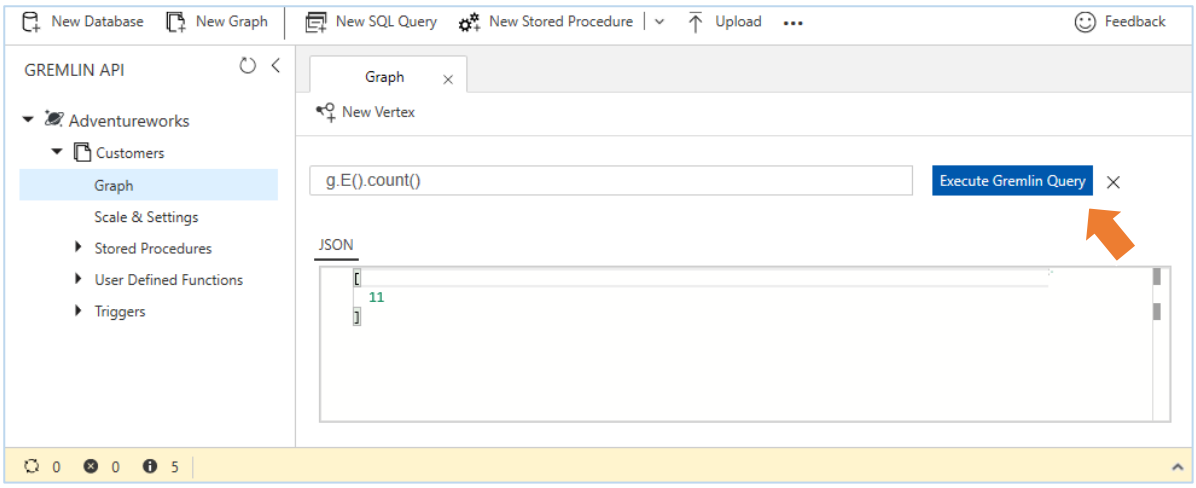
ワン ポイント

count() ステップは、トラバーサルからカウントを返します。グラフには、8 つの Vertex が追加されていることがわかります。

2. ここまでの手順で、グラフに追加した Edge の数をカウントするために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

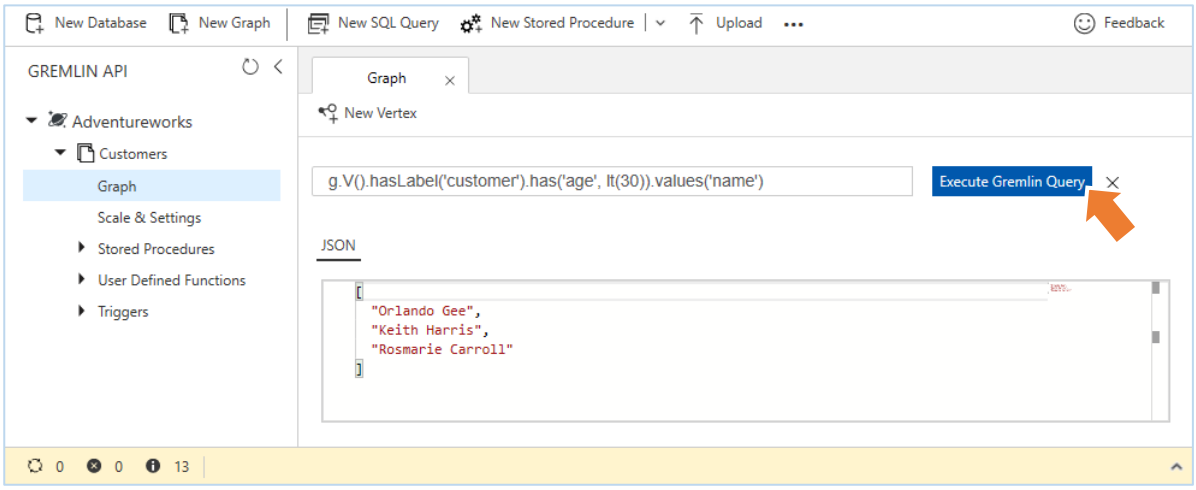
```
g.E().count()
```

2. Azure ポータルからのデータベースの作成と操作



3. customer ラベルの Vertex のプロパティを使用したフィルター処理を実行するため、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().hasLabel('customer').has('age', lt(30)).values('name')
```



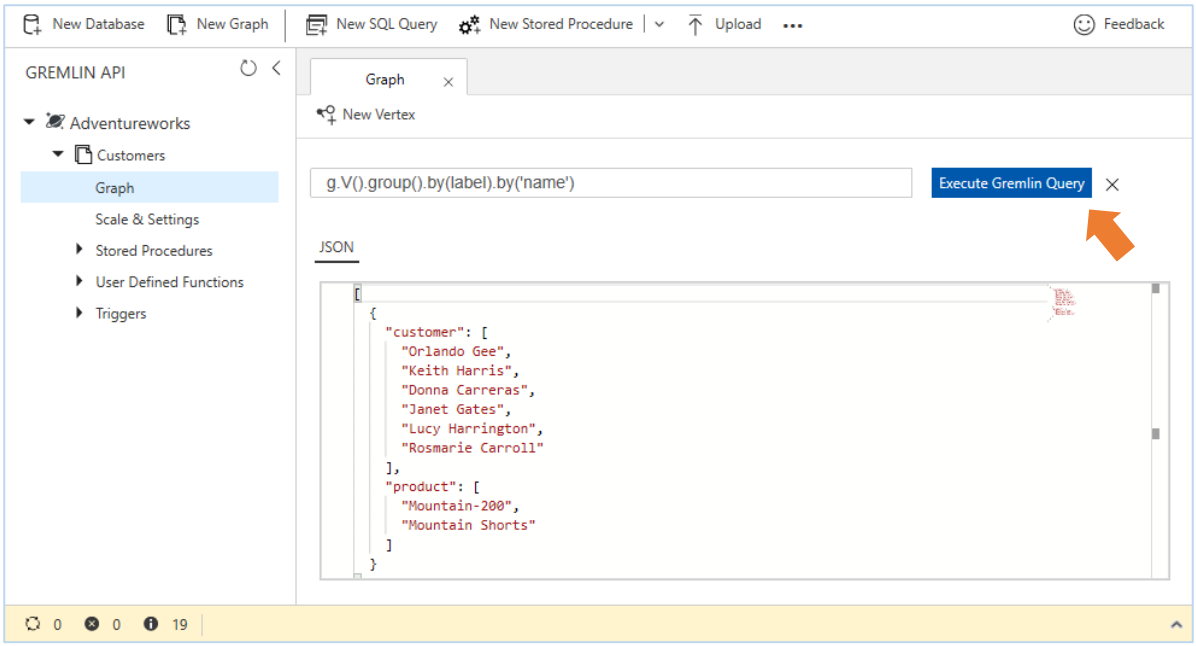
ワン ポイント

hasLabel()、hasId()、および has() ステップは、Vertex や Edge のプロパティのフィルター処理に使用します。記述できる演算子には、eq (等しい)、neq (等しくない)、lt (未満)、lte (以下)、gt (より大きい)、gte (以上) があります。また、between を記述して範囲を設定することもできます。ここで実行したクエリは、30 歳未満の顧客を表示しています。

4. group() ステップを使用し、全ての Vertex を対象に、ラベルごとのリストを作成するため、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().group().by(label).by('name')
```

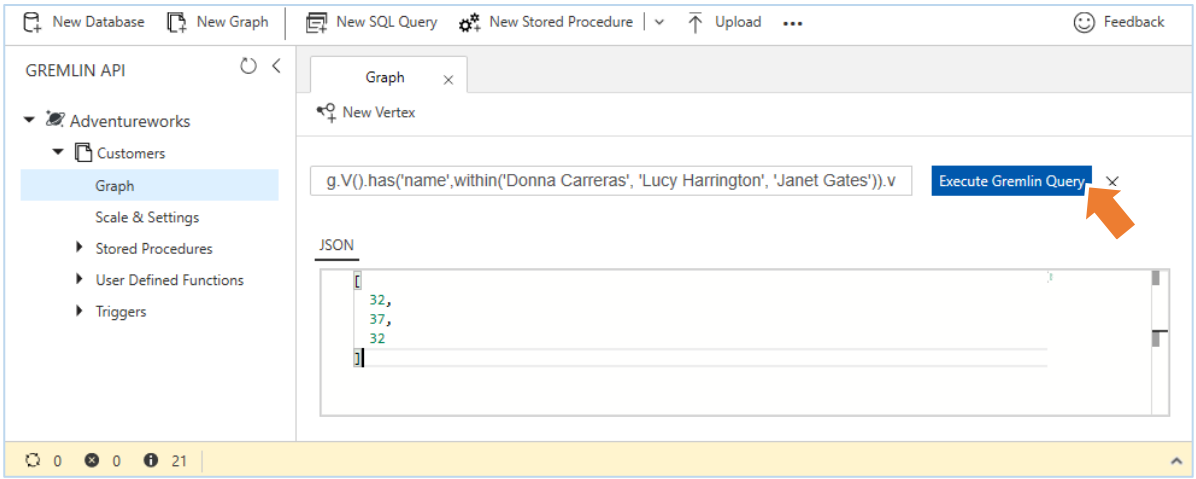
2. Azure ポータルからのデータベースの作成と操作



ワン ポイント
group() ステップは、.by() で指定されたラベルやプロパティに基づいて値をグループ化します。

5. within() ステップを使用し、特定の Vertex を選択して、そのプロパティを表示するため、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().has('name',within('Donna Carreras', 'Lucy Harrington', 'Janet Gates')).values('age')
```

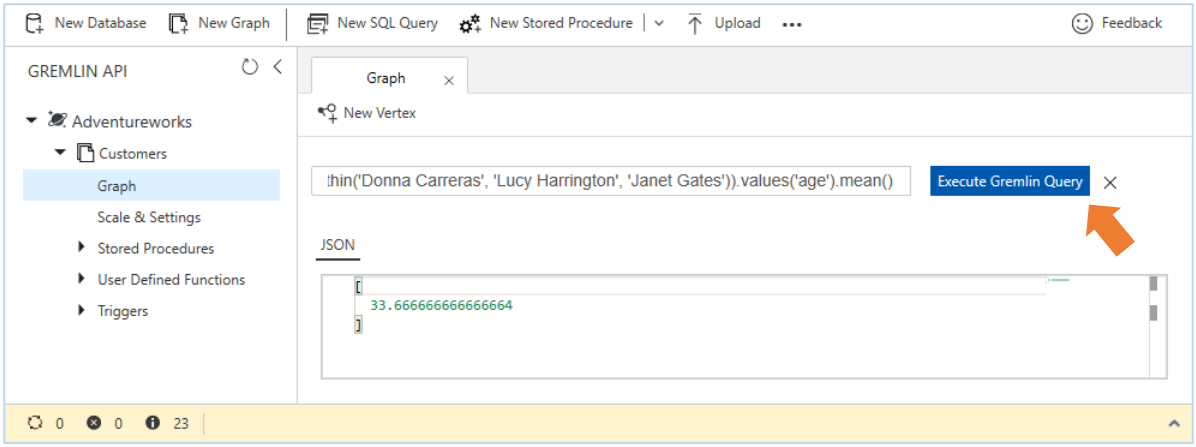


ワン ポイント
within() ステップは、プロパティを指定して複数の Vertex や Edge を列挙することができます。

6. within() ステップを使用した値に対し、集計関数を使用するため、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().has('name',within('Donna Carreras', 'Lucy Harrington', 'Janet Gates')).values('age').mean()
```

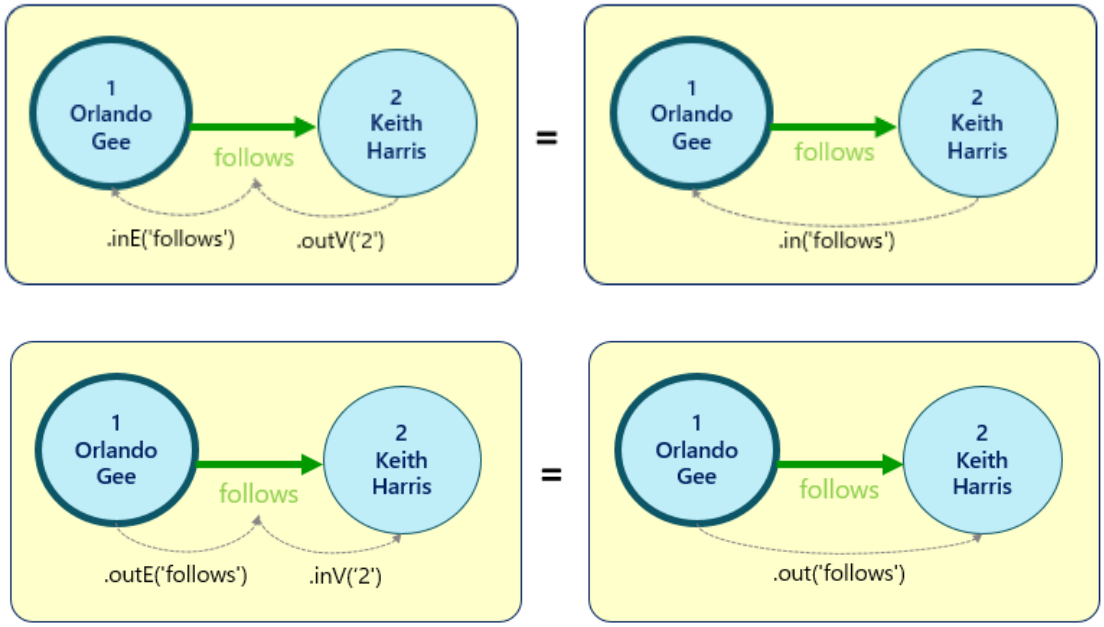
2. Azure ポータルからのデータベースの作成と操作



ワン ポイント
mean() ステップを記述すると平均値を取得できます。その他にも、count()、 sum()、 min()、max() といった集計処理を実行するステップを記述できます。

2.2.5.2 トラバーサル

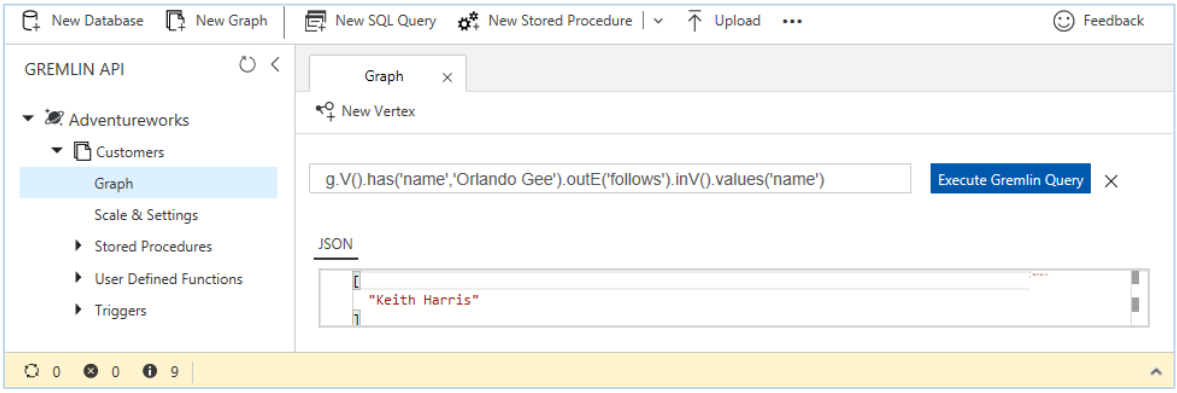
ここでは、in()、out() (inV()、outV()、inE()、outE()) といった方向を指定したトラバーサル操作を実行します。inE() と outE() は、フォーカスしている Vertex から入ってくる方向と出ていく方向にある Edge を示します。inV()、outV() は、フォーカスしている Edge から入ってくる方向と出ていく方向にある Vertex を示します。
inE()、inV() および outE()、outV() の組み合わせは、単一の in() と out() に置き換えることができます。



以降の手順で実際に試してみましょう。

1. Orlando Gee (CustomerID = 1) が、フォローしている顧客を検索するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

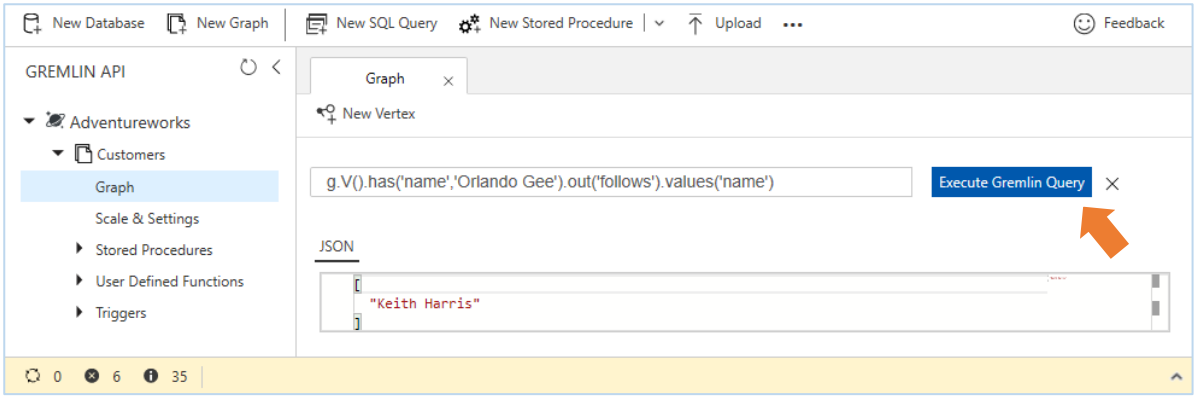
```
g.V().has('name','Orlando Gee').outE('follows').inV().values('name')
```



2. 手順 1 で実行したグラフ クエリは、以下に置き換えることができます。このグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().has('name','Orlando Gee').out('follows').values('name')
```

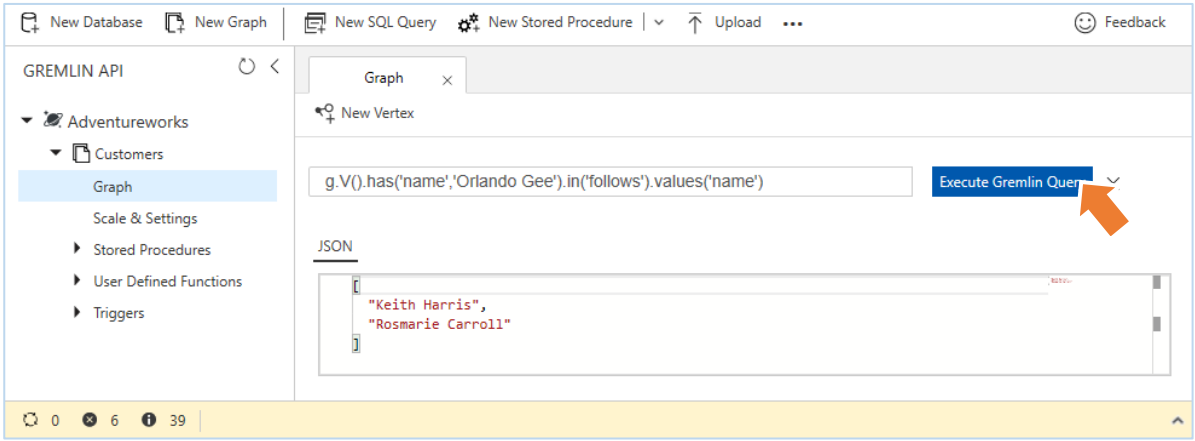
2. Azure ポータルからのデータベースの作成と操作



ワン ポイント
Orlando Gee が、フォローしている顧客は、Keith Harris です。Vertex に対する「.out('follows').values('name')」の記述で、Vertex から出ていく Edge をトラバースされ、その先の Vertex から名前を返します。

3. 逆に、Orlando Gee をフォローしている顧客を検索するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().has('name','Orlando Gee').in('follows').values('name')
```

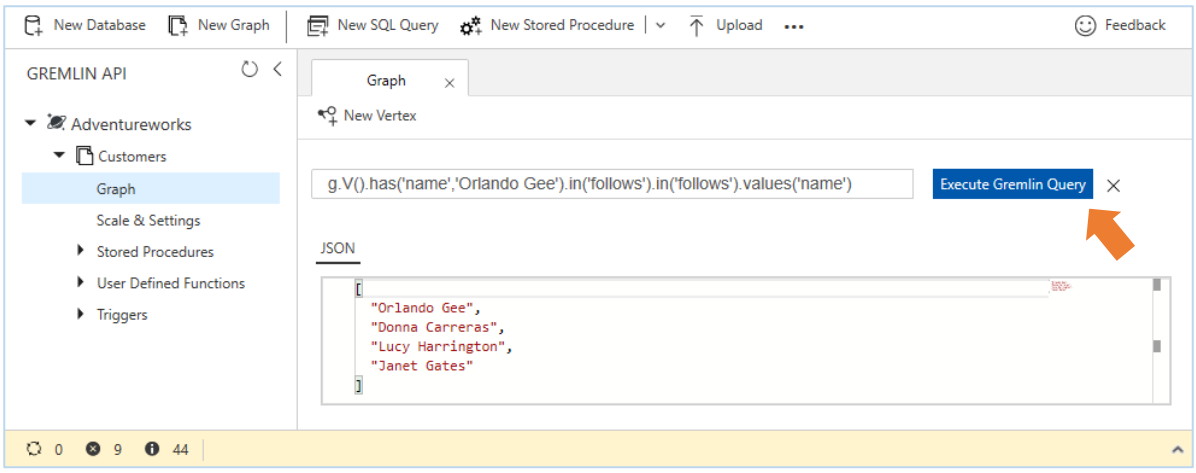


ワン ポイント
Orlando Gee を、フォローしている顧客は、Keith Harris と Rosmarie Carroll の 2 人です。Vertex に対する「.in('follows').values('name')」の記述で、Vertex に入って来る Edge をトラバースされ、その先の Vertex から名前を返します。

4. Orlando Gee (CustomerID = 1) をフォローしている顧客をフォローしている顧客を検索するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().has('name','Orlando Gee').in('follows').in('follows').values('name')
```

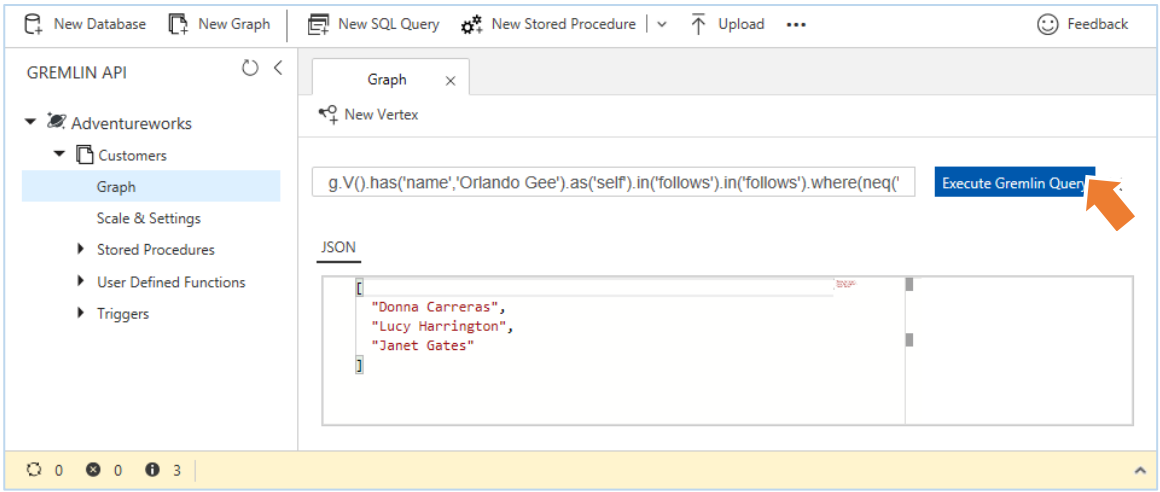
2. Azure ポータルからのデータベースの作成と操作



ワン ポイント
Orlando Gee をフォローしている顧客をフォローしている顧客は、Orlando Gee、Donna Carreras、Lucy Harrington、Janet Gates の 4 人です。Vertex に対する「.in('follows').in('follows').values('name')」の記述で、Vertex に入って来る Edge を 2 つ移動した先の Vertex の名前を返します。フォロワーのフォロワーには、Keith Harris と相互にフォローし合っていたため、Orlando Gee 自身も含まれていますが、これを除外するには、トラバーサルの結果をフィルター処理するための .where() ステップを記述します。

5. Orlando Gee (CustomerID = 1) をフォローしている顧客をフォローしている顧客のトラバーサルから、Orlando Gee 自身を除くために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().has('name','Orlando Gee').as('self').in('follows').in('follows').where(neq('self')).values('name')
```

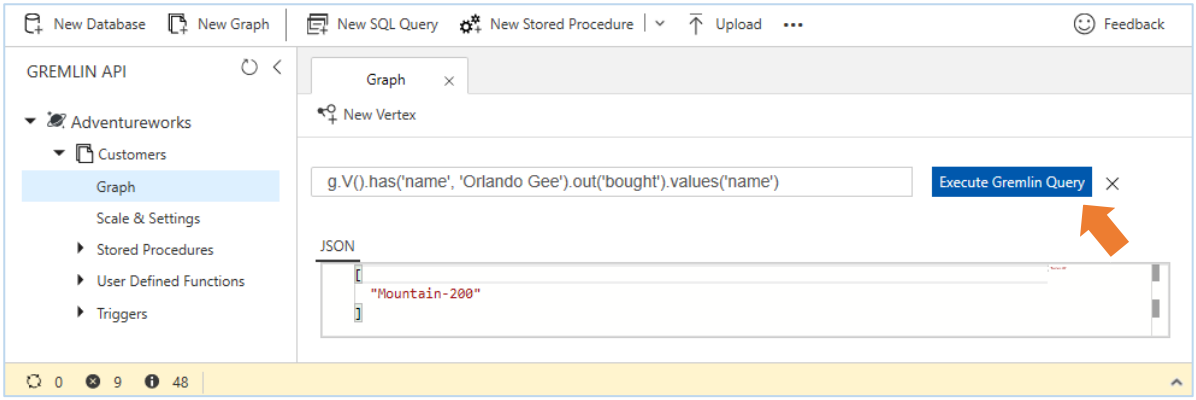


ワン ポイント
Orlando Gee 自身は、除外されました。.as() ステップは、ステップからの出力に変数を割り当てることができます。

6. bought ラベルの Edge を使用して、Orlando Gee が購入した商品の名前をトラバーサルします。以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().has('name', 'Orlando Gee').out('bought').values('name')
```

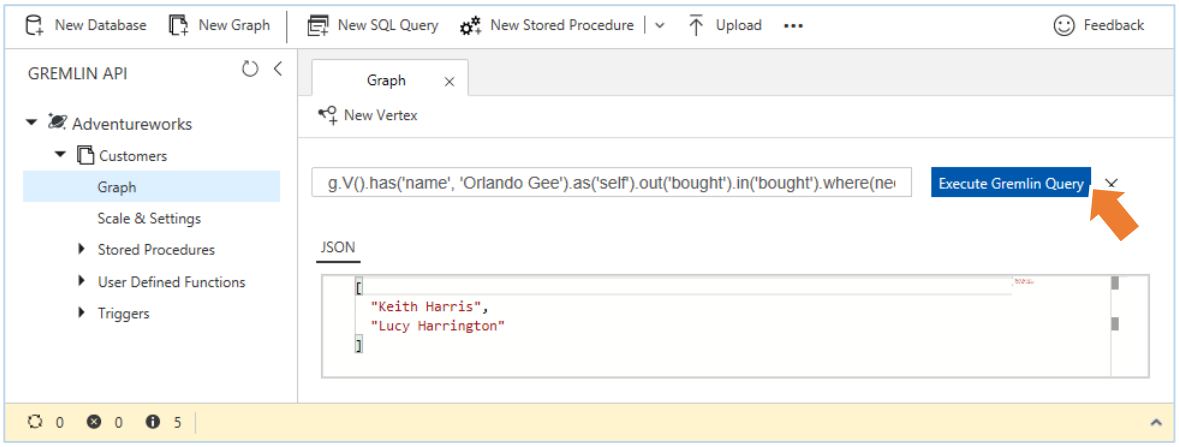
2. Azure ポータルからのデータベースの作成と操作



ワン ポイント
Orlando Gee が、Mountain-200 を購入していることがわかります。

7. 次に、Orlando Gee が購入した Mountain-200 を購入している他の顧客の名前をトラバーサルするため、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().has('name', 'Orlando Gee').as('self').out('bought').in('bought').where(neq('self')).values('name')
```

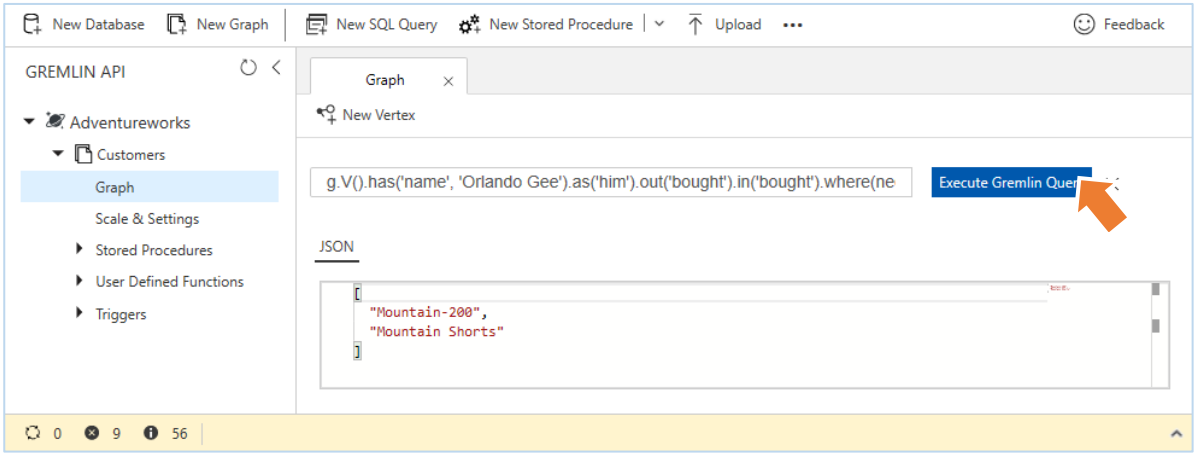


ワン ポイント

Keith Harris と Lucy Harrington も Mountain-200 を購入していることがわかります。.where() ステップを記述して、Orlando Gee 自身が表示されないようにフィルターしています。

8. さらに、Orlando Gee が購入した Mountain-200 を購入している他の顧客が購入している商品の名前をトラバーサルするため、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().has('name', 'Orlando Gee').as('self').
      out('bought').
      in('bought').where(neq('self')).
      out('bought').
      dedup().values('name')
```



ワン ポイント

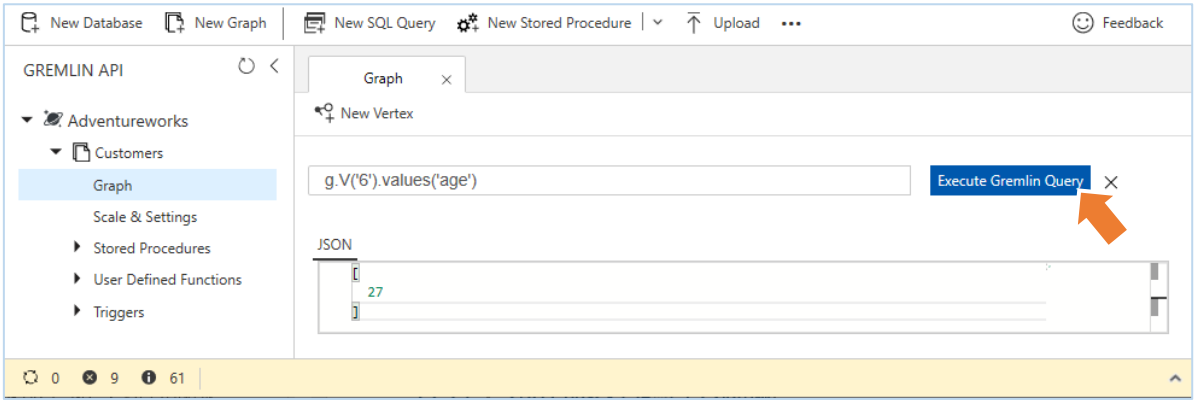
Mountain-200 と Mountain Shorts を購入していることがわかります。dedup() ステップは、重複する値を削除して返します。

2.2.5.3 変更と削除の操作

ここでは、プロパティの変更や Vertex や Edge を削除する方法を確認します。Vertex や Edge の削除では、誤って必要なものを削除しないようにするため、ターゲットの id を指定して、削除することをお勧めします。

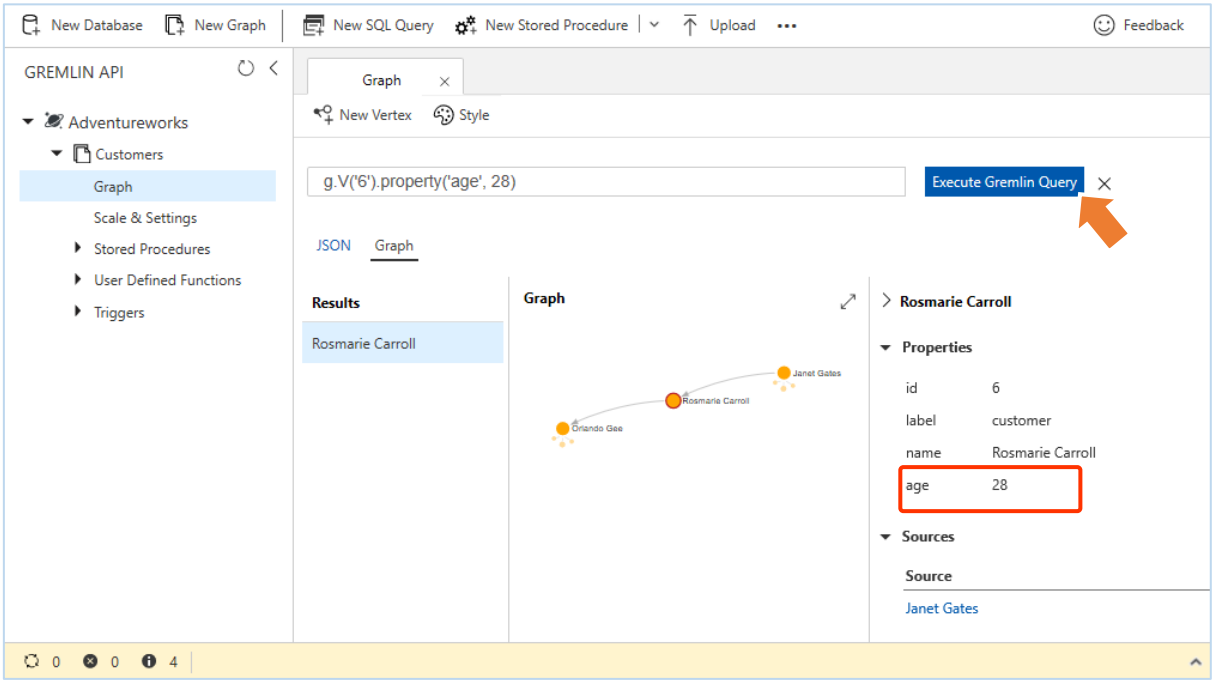
1. Rosmarie Carroll (CustomerID = 6) の年齢を確認するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V('6').values('age')
```



2. Rosmarie Carroll の年齢を 27 から 28 に変更するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V('6').property('age', 28)
```



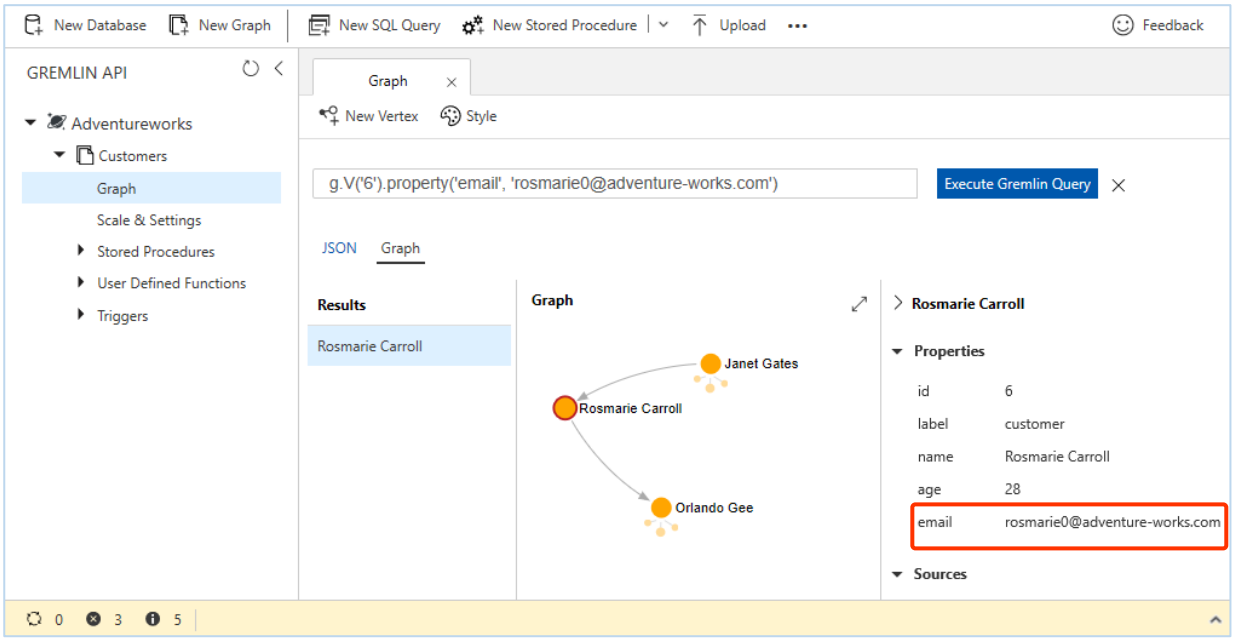
ワン ポイント

Rosmarie Carroll の年齢は、28 に変更されました。プロパティを変更するには、対象となる Vertex や Edge をフィルターして、.property() でキー/バリュー値を再設定します。この方法で、新しいプロパティを追加することができます。

2. Azure ポータルからのデータベースの作成と操作

3. Rosmarie Carroll に email プロパティを追加するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

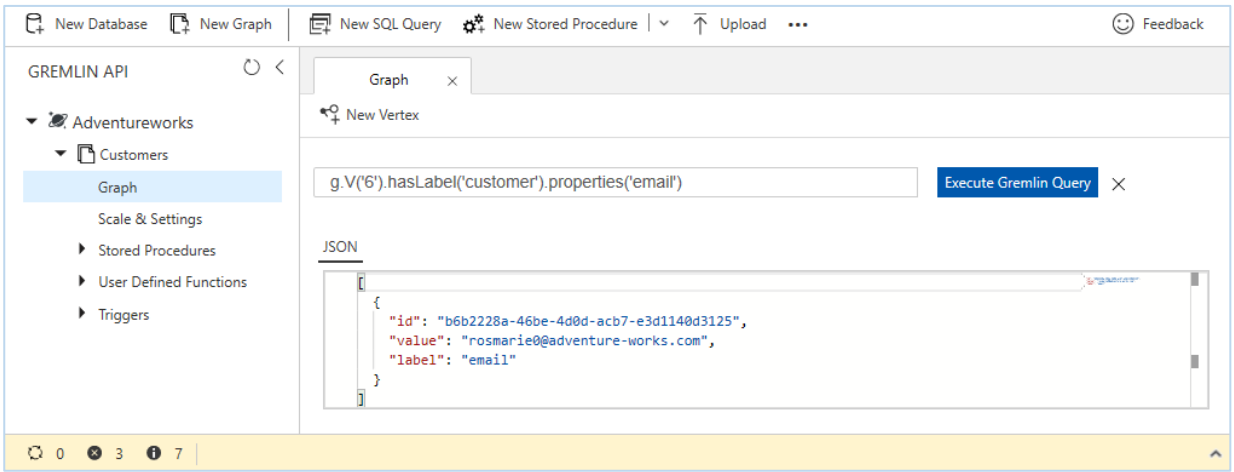
```
g.V('6').property('email', 'rosmarie0@adventure-works.com')
```



ワン ポイント
Rosmarie Carroll に email プロパティが追加されました。

4. Rosmarie Carroll の email プロパティを検索するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

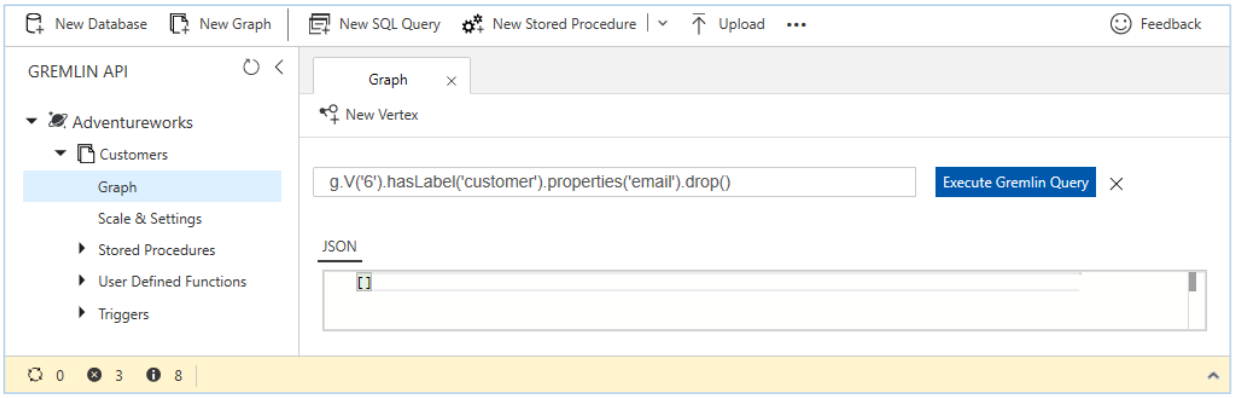
```
g.V('6').hasLabel('customer').properties('email')
```



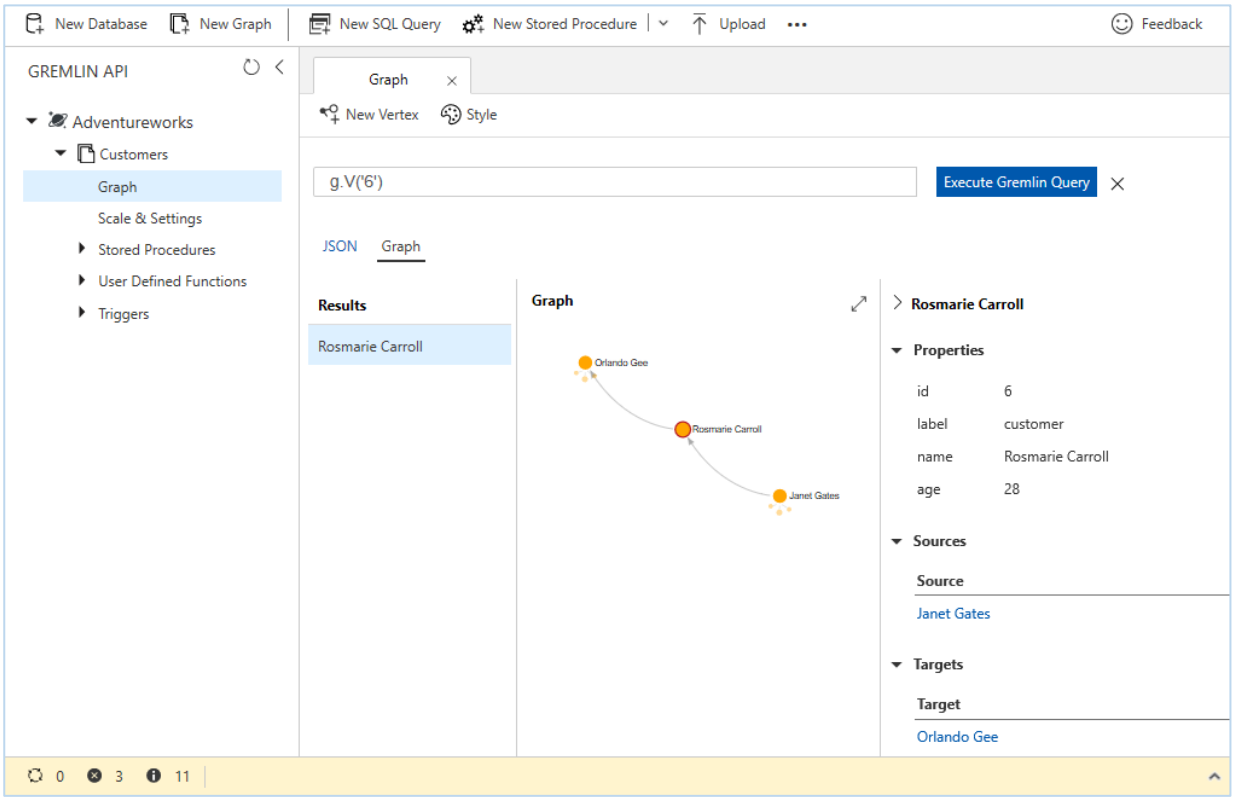
ワン ポイント
プロパティの削除は、対象となるプロパティをフィルターして drop() ステップを使用します。

5. Rosmarie Carroll の email プロパティを削除するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V('6').hasLabel('customer').properties('email').drop()
```



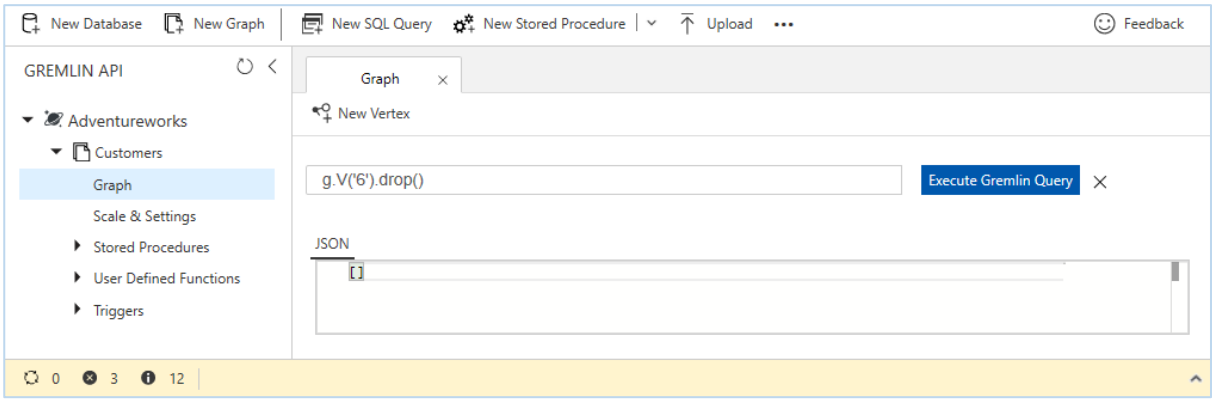
6. Rosmarie Carroll (id=6) のプロパティを確認するために、グラフ クエリとして「**g.V('6')**」と入力して、[Execute Gremlin Query] をクリックします。



ワン ポイント
Rosmarie Carroll から email プロパティが削除されました。Vertex に対して直接 drop() ステップを使用すると Vertex そのものが削除されます。

7. Rosmarie Carroll (CustomerID = 6) を削除するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

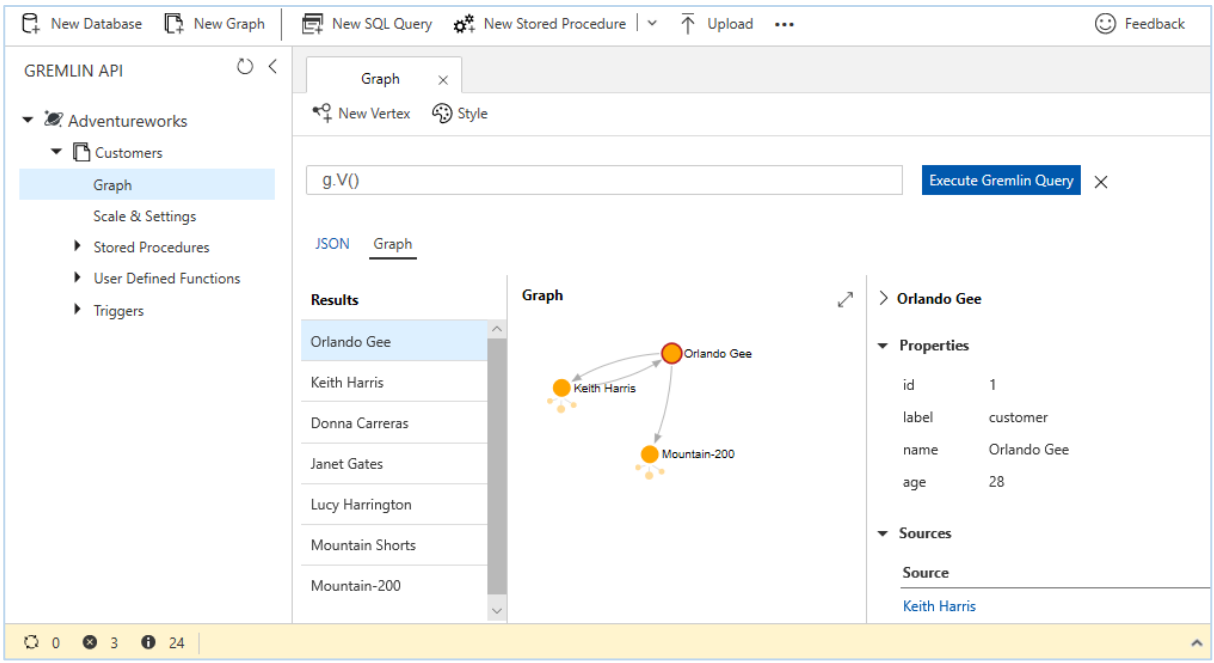
```
g.V('6').drop()
```



ワン ポイント

複数の Vertex を削除する場合、「g.V('3','4','5').drop()」のように .V() に id を列挙することができます。
また、「g.V().drop()」のように、.V() に引数を指定しないと、すべての Vertex と Edge が削除されることに注意してください。

8. 結果を確認するために、[Data Explorer] ブレードで、「**g.V()**」と入力して、[Execute Gremlin Query] をクリックしてグラフ クエリを実行します。

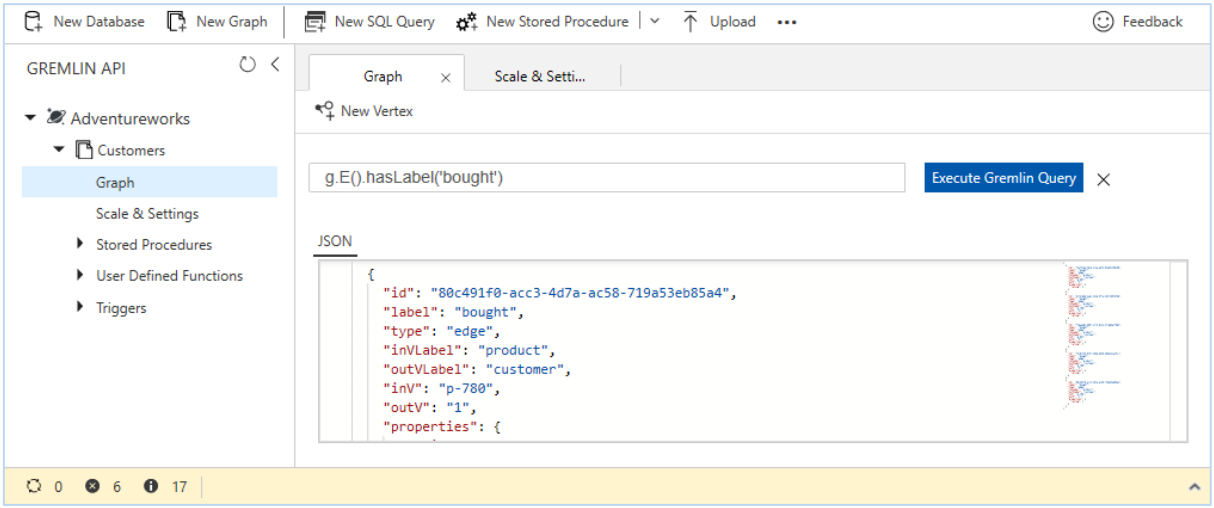


ワン ポイント

Rosmarie Carroll (CustomerID = 6) が削除されたことがわかります。また、Orlando Gee (id=1) は、Mountain-200 を購入していることを確認しておきます。

9. Edge の情報を取得するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.E().hasLabel('bought')
```

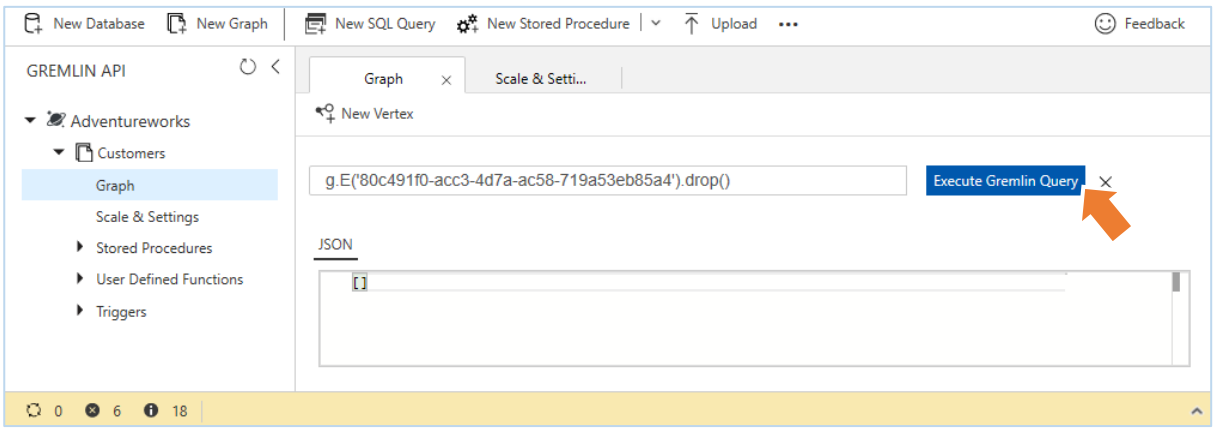


ワン ポイント

bought ラベルの Edge で inV が「p-780」で outV が「1」の id を確認します。上記の画面では、「80c491f0-acc3-4d7a-ac58-719a53eb85a4」です。

10. Orlando Gee (id=1) から Mountain-200 (id=780) への bought ラベルの Edge を削除するために、以下のグラフ クエリの .E() の引数に、前の手順で確認した id 値を入力して、[Execute Gremlin Query] をクリックします。

```
g.E('80c491f0-acc3-4d7a-ac58-719a53eb85a4').drop()
```

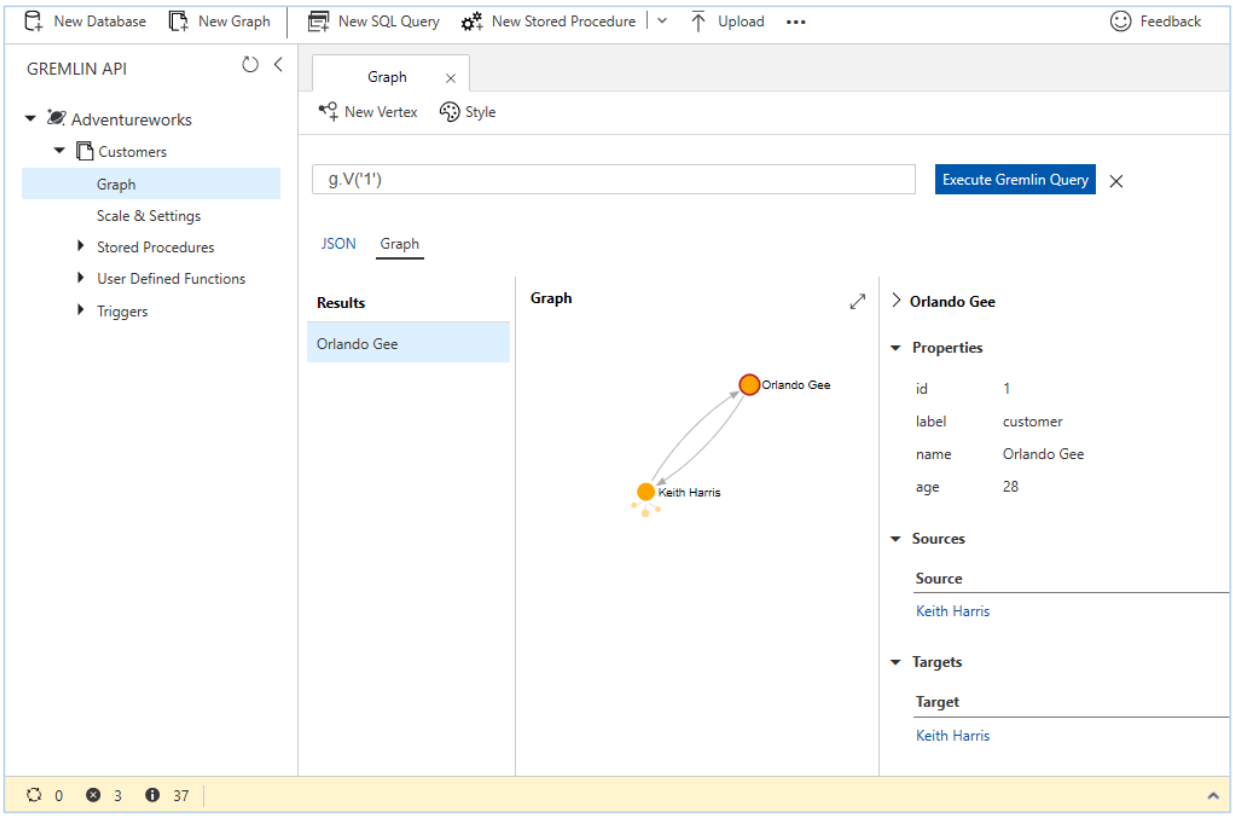


ワン ポイント

drop() ステップを使用して、Vertex、Edge、プロパティの削除を実行できます。

2. Azure ポータルからのデータベースの作成と操作

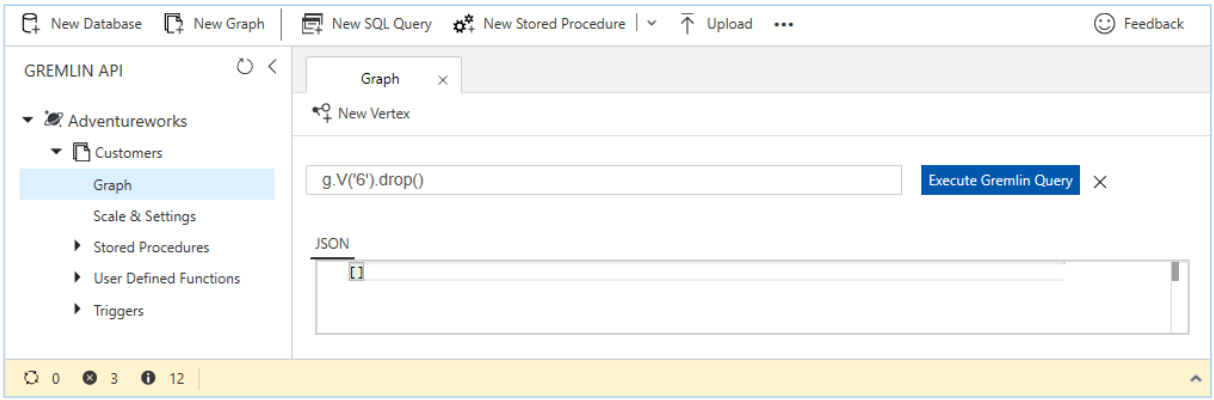
11. Orlando Gee (id=1) を中心としたエンティティのつながりを確認するために、グラフ クエリとして「**g.V('1')**」と入力して、[Execute Gremlin Query] をクリックします。



ワン ポイント
Orlando Gee (id=1) から Mountain-200 への Edge は存在しないことを確認します。

12. Edge からすべての Vertex と Edge を削除するために、以下のグラフ クエリを入力して、[Execute Gremlin Query] をクリックします。

```
g.V().drop()
```

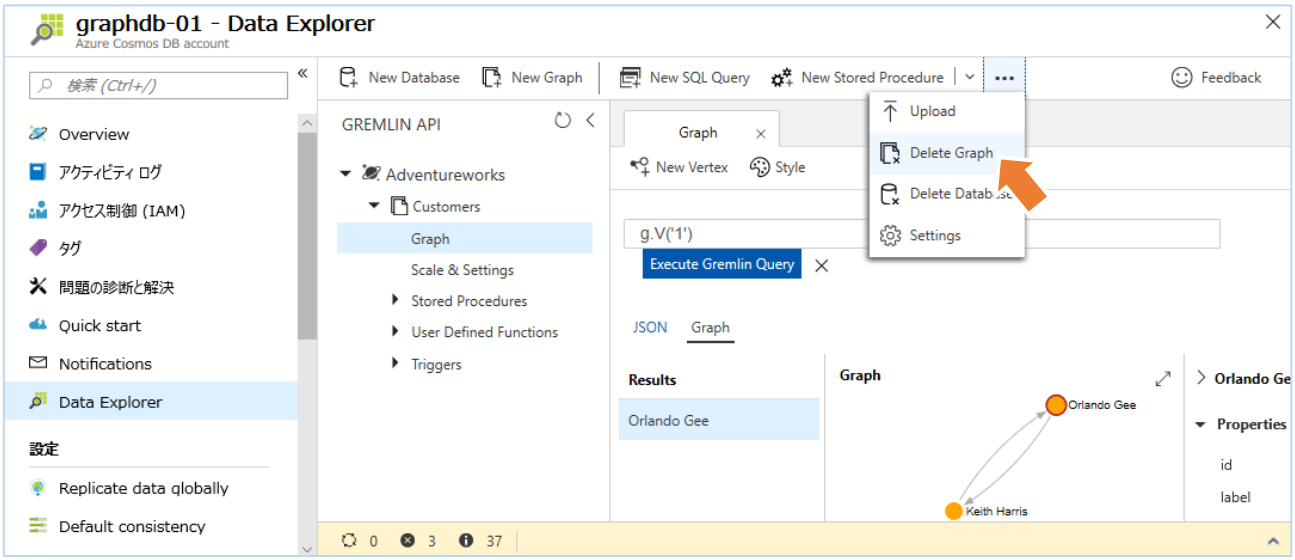


ワン ポイント
「g.V().drop()」を実行すると、Vertex と Edge がすべて削除されます。ただし、コンテナーである Customers グラフを削除しないと課金が継続していることに注意してください。

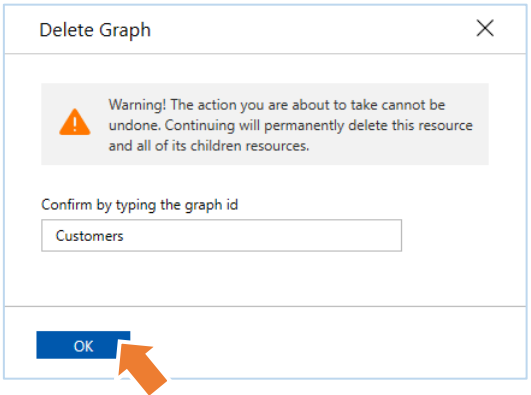
2.2.6 グラフの削除（オプション）

作成したグラフを削除します。引き続き、3 章の手順を実行する場合は、このセクションを実行しないでください。3 章では、空の Customers グラフを使用して、2 章で追加した Vertex と Edge をアプリケーション コードで作成します。同じ名前の Customers グラフをコンテナーとして使用しますので、3 章の手順を実行する場合は、このセクションはスキップして、3 章を開始してください。

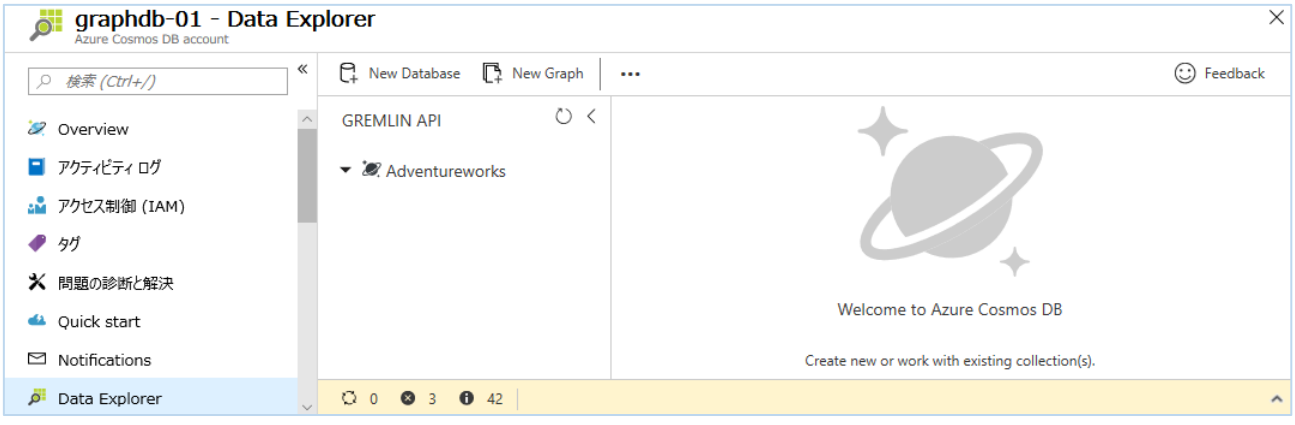
1. Customers グラフを削除するために、[Data Explorer] ブレードで [Delete Graph] をクリックします。



2. [Delete Graph] が表示されるので、[Confirm by typing the graph id] に「**Customers**」と入力して [OK] をクリックします。



3. データベースからコンテナが削除されたことを確認します。



3. Gremlin API SDK を使用する .NET コンソール アプリケーションの作成

2018 年 12 月時点では、Azure Cosmos DB の Gremlin API を使用するアプリの開発に使用できるドライバーは下表の種類が提供されています。

ドライバー	ダウンロード	サポートされている コネクタ バージョン	説明
Gremlin.Net	Nuget Gallery	3.4.0	C # で Gremlin クライアントを実装できます。.NET Standard をターゲットとしているため、.NET Framework や .NET Core で使用できます。TinkerPop3 の Gremlin.Net でドキュメントが参照できます。
Gremlin for The Java	MVN Repository	3.2.0 以降	Java Development Kit (JDK) 1.7 以降の環境で、Gremlin クライアントを実装できます。実装手順は、「Azure Cosmos DB: グラフ データベースを Java と Azure Portal で作成する」を参照してください。
Gremlin-Javascript	Node Package Manager (NPM)	2.6.0	Node.js (v0.10.29 以降) ランタイムを実行できる OS 環境で Gremlin クライアントを実装できます。TinkerPop3 の Gremlin-JavaScript でドキュメントが参照できます。実装手順は、「Azure Cosmos DB: Gremlin API を使用した Node.js アプリケーションの構築」を参照してください。
Gremlin-Python	pip package manager	3.2.7	Python バージョン v3.5 以降の環境で、Gremlin クライアントを実装できます。 TinkerPop3 の Gremlin-Python でドキュメントが参照できます。実装手順は、「Azure Cosmos DB: Python と Azure Portal を使用してグラフ データベースを作成する」を参照してください。
gremlin-server client for php	Packagist	3.1.0	PHP 5.6 以降で Gremlin クライアントを実装できます。実装手順は、「Azure Cosmos DB: グラフ データベースを PHP と Azure Portal で作成する」を参照してください。

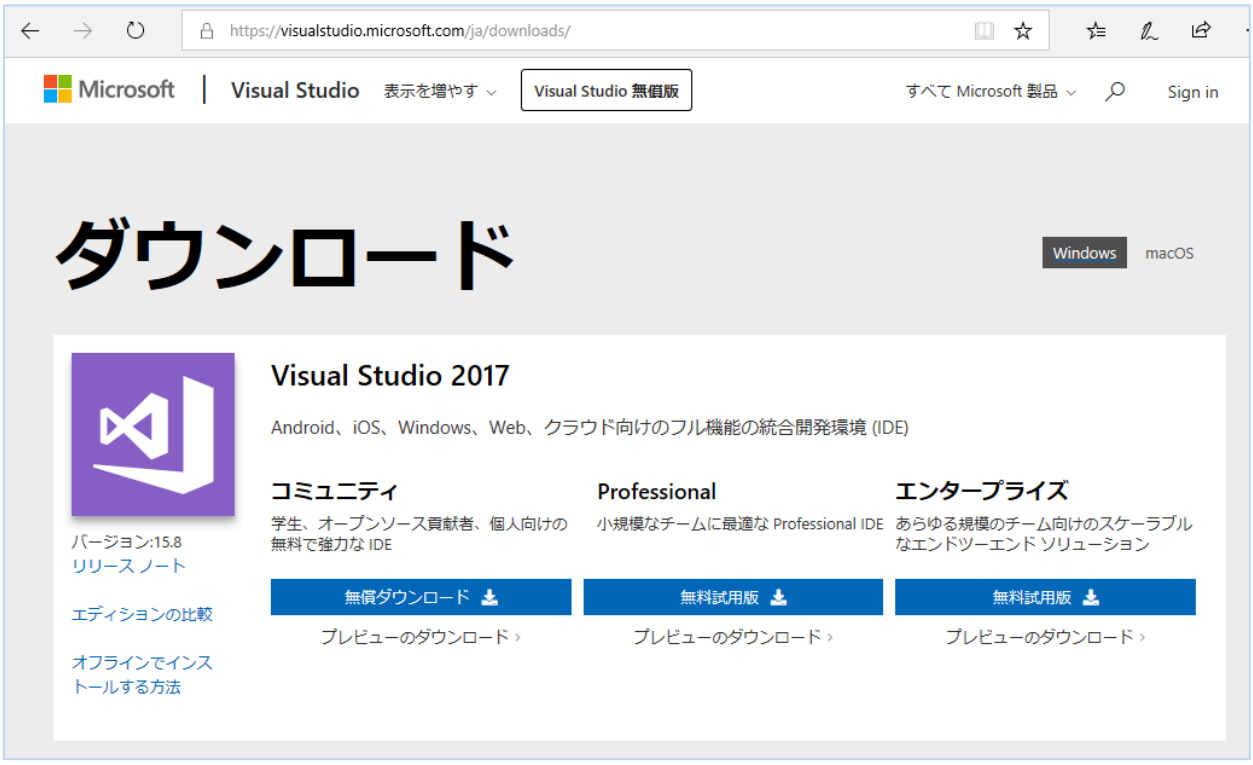
本自習書では、Gremlin.Net API を使用する C# のコードを記述して、Azure Cosmos DB アカウントに接続する方法を学習します。Gremlin.Net API は、すべての .NET 実装で使用できるようにすることを目的とした .NET Standard をターゲットとしているため、.NET Framework や .NET Core でも利用することができ、Windows 以外のオペレーティング システムでも利用可能です。

サポートされているドライバーの最新情報は、「Azure Cosmos DB の概要: Gremlin API」を参照してください。

3.1 Visual Studo 2017 のインストール

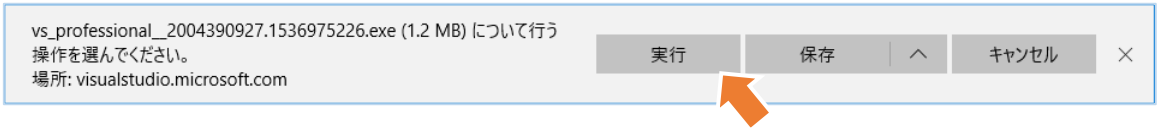
自習書の実行環境に Visual Studio 2017 をインストールしていない場合、この手順を実行することで、Visual Studio 2017 をインストールできます。Visual Studio 2017 は、https://visualstudio.microsoft.com/ja/downloads から入手できます。

1. Web ブラウザーで https://visualstudio.microsoft.com/ja/downloads にアクセスして、表示されるページで、使用する Visual Studo のダウンロード ボタンをクリックします。

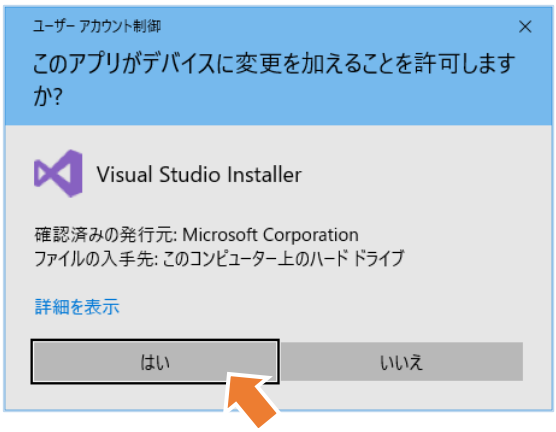


ワン ポイント
Mac OS 版の Visual Studio を使用する場合、[ダウンロード] の右側の [macOS] をクリックしてください。

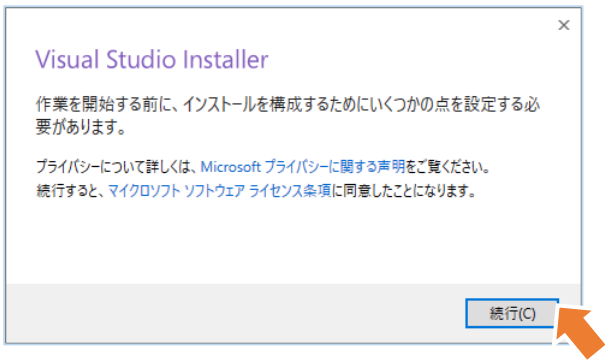
2. Web ブラウザーの下部にメッセージが表示されたら、[実行] をクリックします。



3. [ユーザー アカウントの制御] が表示されたら、[はい] をクリックします。



4. 次のメッセージ ボックスで、リンクをクリックして「Microsoft のプライバシーに関する声明」と「マイクロソフト ソフトウェア ライセンス条項」を確認し、よろしければ [続行] をクリックして インストールを開始します。

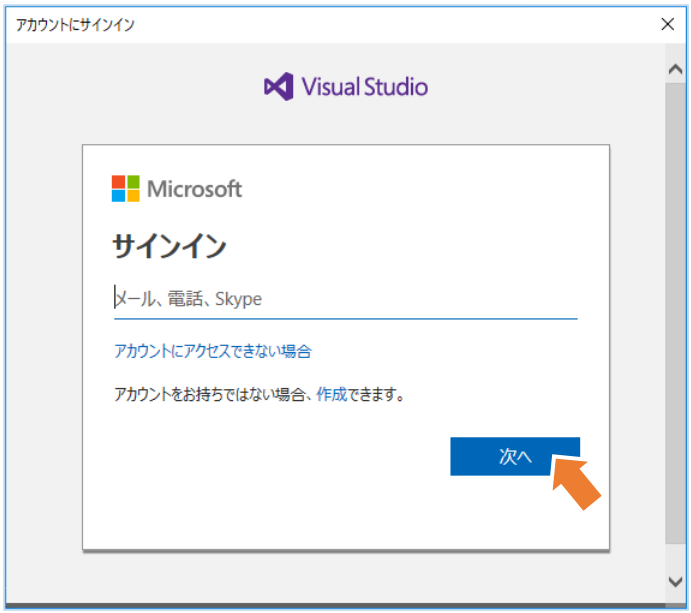


5. ワークロードを選択するページが開くので、「.NET デスクトップ開発」と「Azure の開発」を選択して [インストール] ボタンをクリックします。

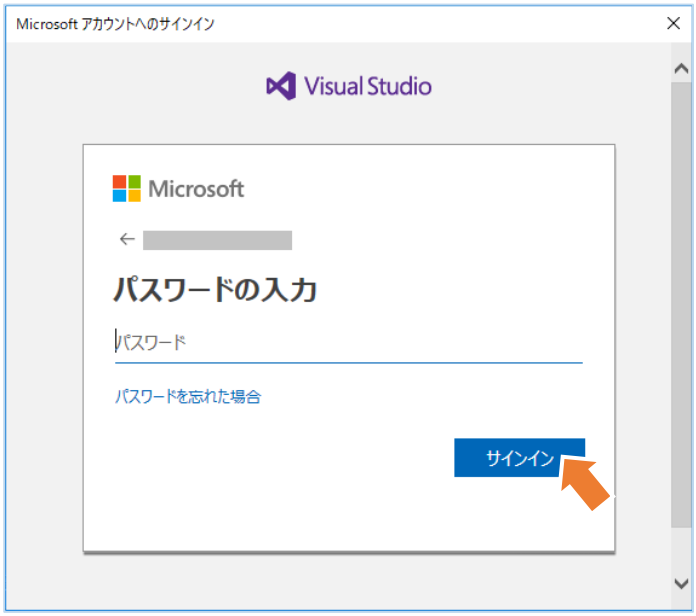
> **ワン ポイント**
>
> C# で Azure Cosmos DB に接続する .NET コンソール アプリを作成するために必要最小限のワークロードを選択しています。

6. インストール完了後に、[ようこそ] が表示されるので、[サインイン] ボタンをクリックします。

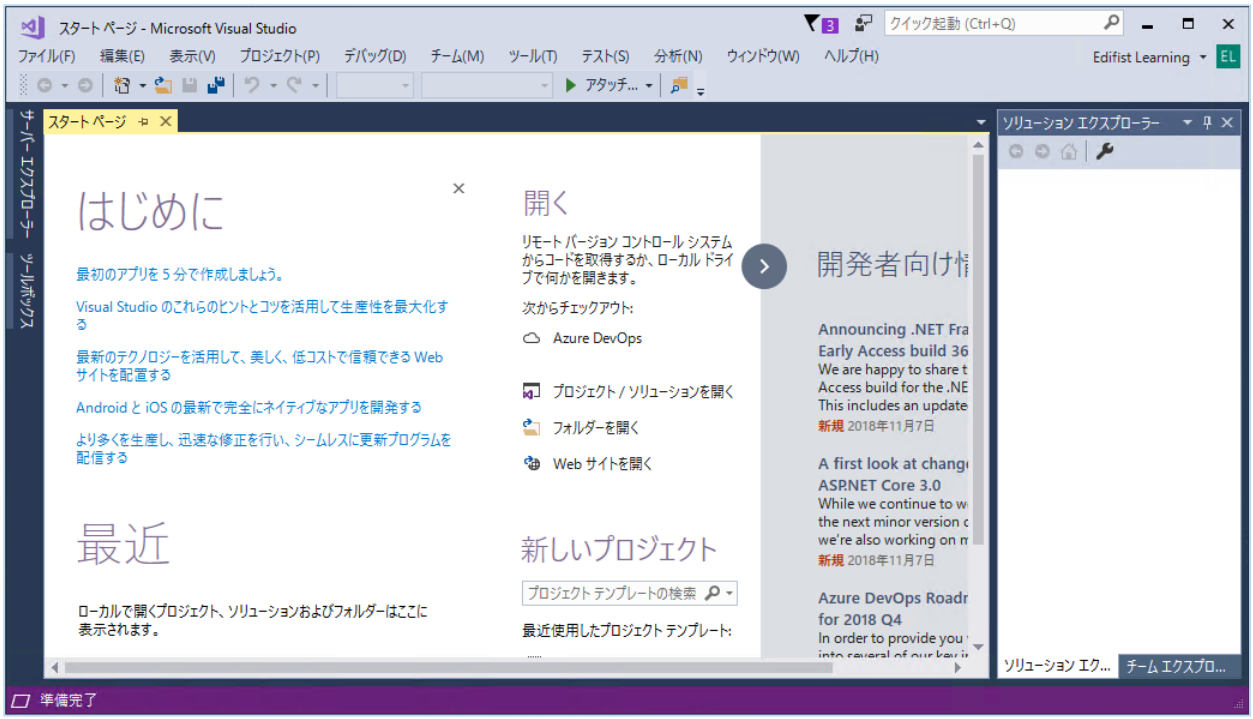
7. [サインイン] ページでは、前の章の手順を実行したときに使用した、Azure サブスクリプションが関連付けられたアカウントを入力して、[次へ] ボタンをクリックします。



8. Azure アカウントのパスワードを入力して [サインイン] ボタンをクリックします。



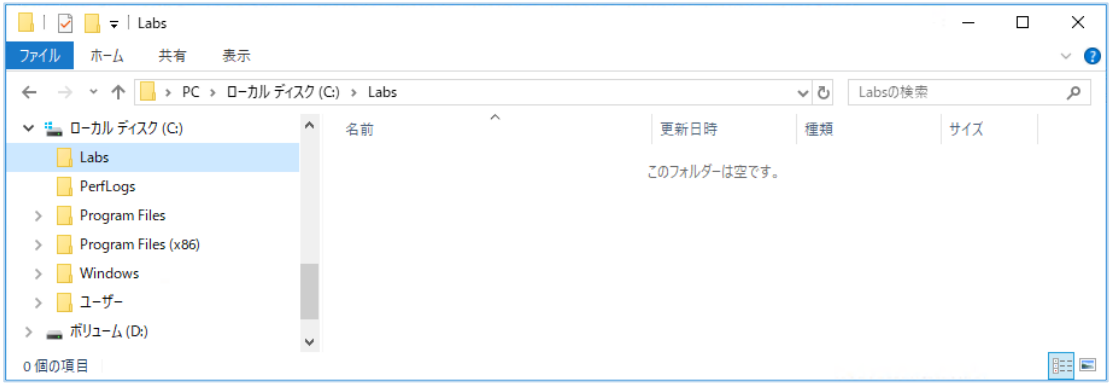
9. Visual Studo が起動することを確認します。



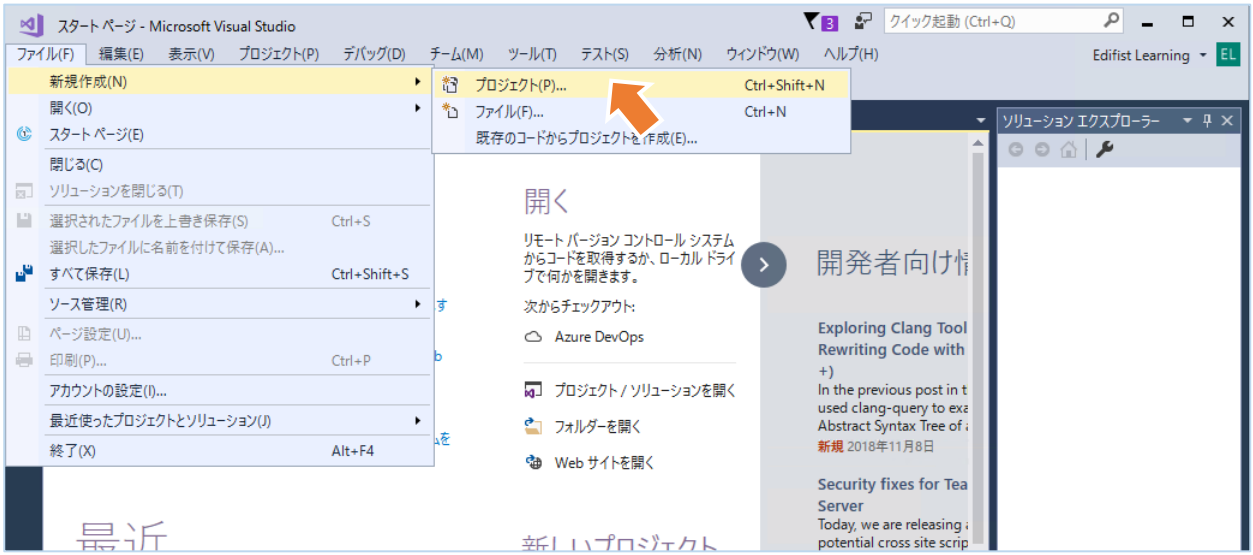
3.2 コンソール アプリのプロジェクト作成と NuGet パッケージのダウンロード

ここでは最初に、コンソール アプリのプロジェクトを作成します。Azure Cosmos DB アカウントに接続し、グラフに対する操作を行うのに必要なクラスが含まれる **Gremlin.Net** NuGet パッケージをダウンロードし、コンソール アプリのプロジェクトから参照できるように構成します。

1. Windows エクスプローラーで、自習用のディスク ドライブに、「Labs」という名前のフォルダーを作成します。



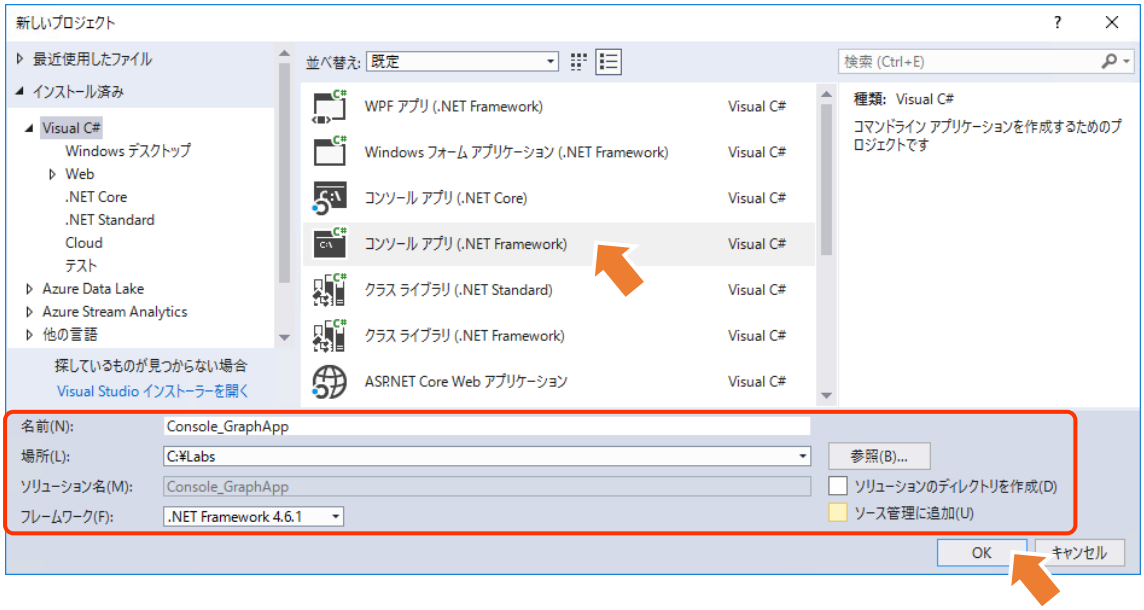
2. Visual Studio の [ファイル] メニューから [新規作成] ⇒ [プロジェクト] を選択します。



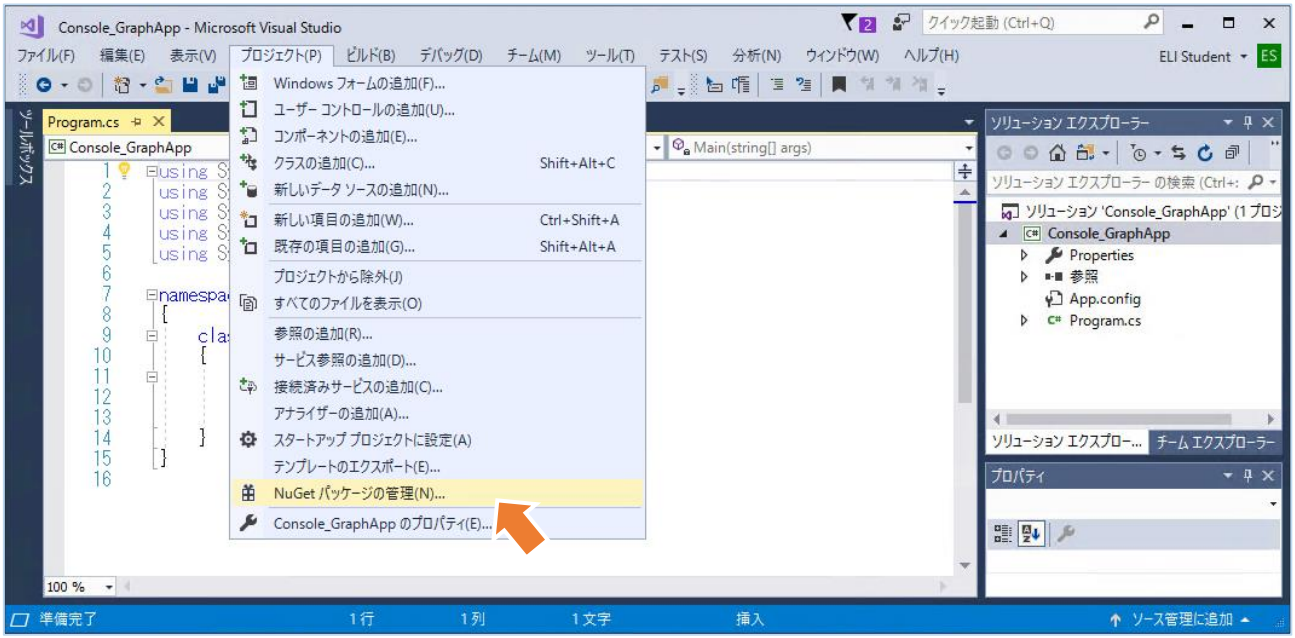
3. [新しいプロジェクト] が開くので、[Visual C#] ノードの下の [コンソール アプリ (.NET Framework)] テンプレートを選択して、下表に従いプロジェクトを構成した後、[OK] ボタンをクリックします。

項目	設定値
名前	Console_GraphApp
場所	C:¥Labs
フレームワーク	.NET Framework 4.6.1
ソリューションのディレクトリを作成	選択を外す
ソース管理に追加	選択を外す

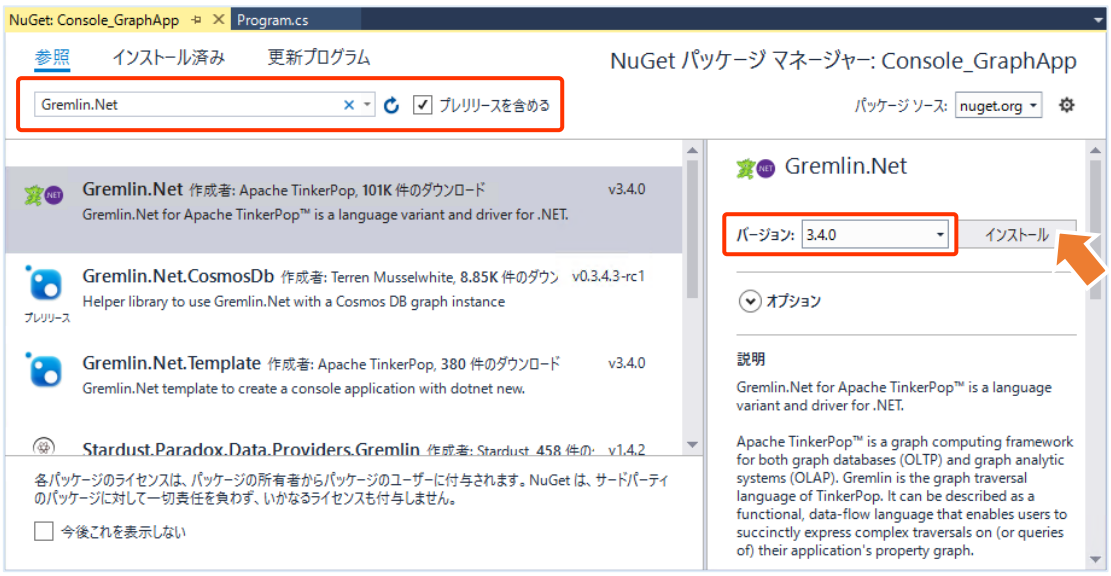
3.gremlin API SDK を使用する .NET コンソール アプリケーションの作成



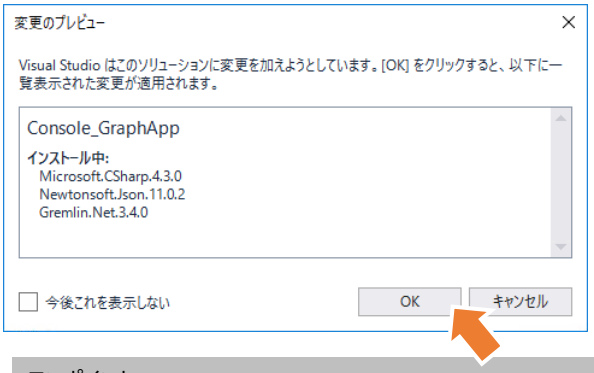
4. 必要な NuGet パッケージを取得するために、[プロジェクト] メニューを開き [NuGet パッケージの管理] をクリックします。



5. NuGet パッケージ マネージャーの [参照] タブを選択し、[プレリリースを含める] チェックボックスを選択した後、[検索] ボックスに「**Gremlin.Net**」と入力して、表示される Microsoft.Azure.Graphs NuGet パッケージを選択し、[バージョン] で「3.4.0」を選択して、[インストール] ボタンをクリックします。

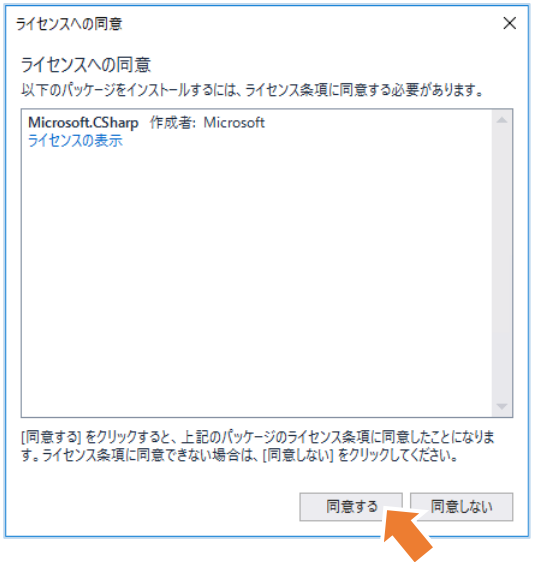


6. 依存関係が収集された後、[変更のプレビュー] が表示されたら、[OK] ボタンをクリックします。



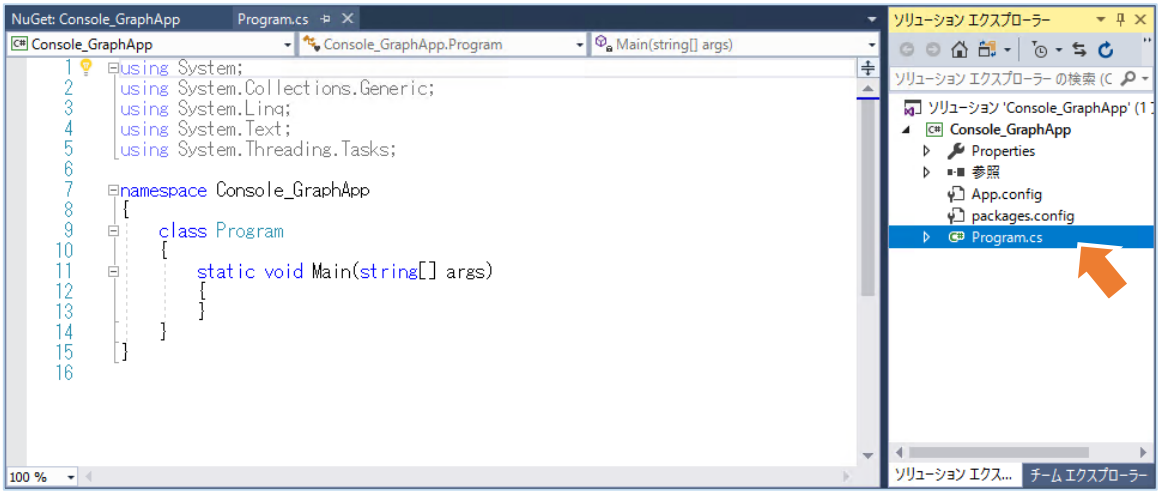
ワンポイント
選択した NuGet に含まれるパッケージが、ソリューションにインストールされます。

7. この後、[ライセンスへの同意] が表示されたら、[同意する] ボタンをクリックします。



3.gremlin API SDK を使用する .NET コンソール アプリケーションの作成

8. ソリューション エクスプローラーのツリーで Program.cs ファイルをクリックして、エディターで開きます。



9. Program.cs ファイルの先頭の using ディレクティブの以下の 2 行は使用しないため、削除します。

```
using System.Linq;
using System.Text;
```

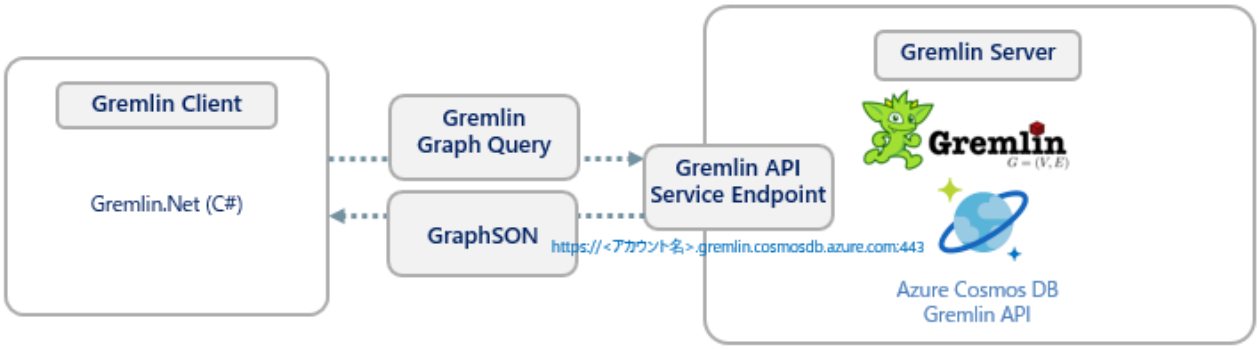
10. using ディレクティブのブロックにクラス参照のために、次のコードを追加します。

```
using System.Configuration;

using Gremlin.Net.Driver;
using Gremlin.Net.Driver.Exceptions;
using Gremlin.Net.Structure.IO.GraphSON;
using Newtonsoft.Json;
```

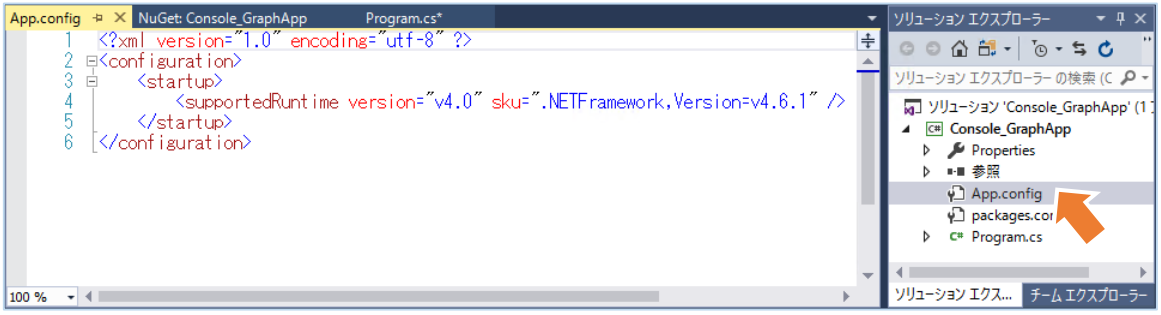
ワンポイント
「System.Configuration」は、App.config 構成ファイルから構成情報を取得するために使用します。

3.3 Azure Cosmos DB アカウントとグラフへの接続

Azure Cosmos DB アカウントへの接続のために、App.Config に接続文字列を記述し、GremlinServer クラスをインスタンス化します。GremlinServer インスタンスから GremlinClient インスタンスを作成して、その SubmitAsync メソッドで、GremlinServer にグラフ クエリを送信して非同期実行することができます。



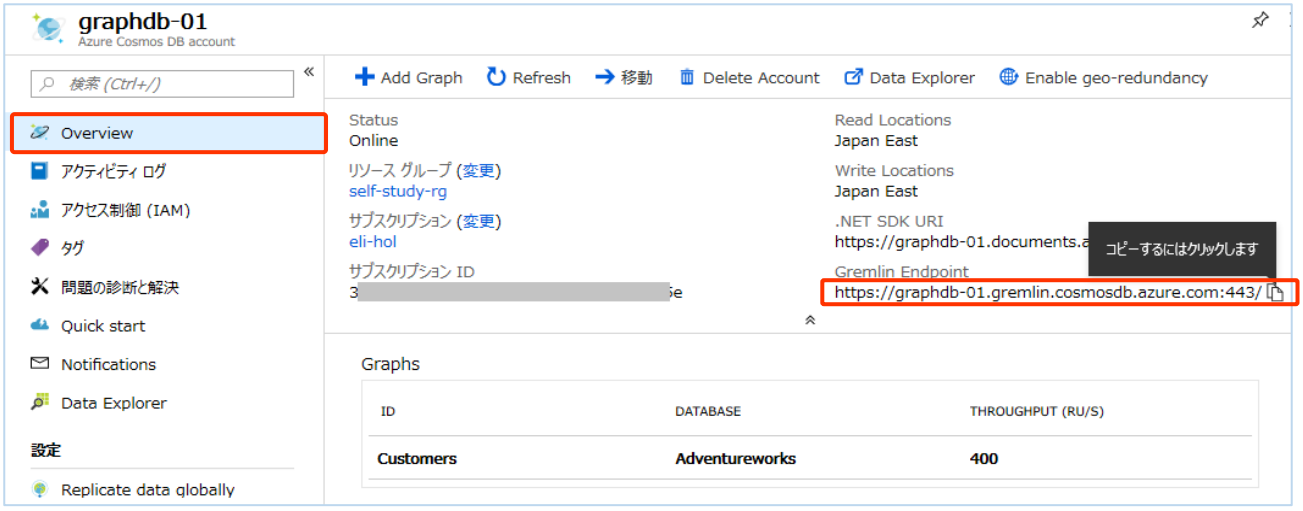
1. ソリューション エクスプローラーのツリーで App.config ファイルをクリックして、エディターで開きます。



2. <configuration><startup> 要素内に次の <appSettings> 要素を追加します。

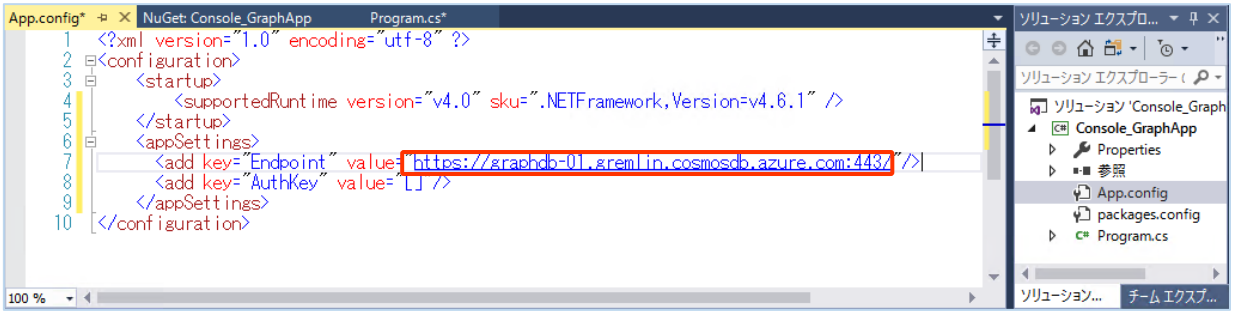
```
<appSettings>
    <add key="Endpoint" value="[URI]"/>
    <add key="AuthKey" value="[KEY]"/>
</appSettings>
```

3. Azure ポータルに戻り、アプリケーション コードから接続する Azure Cosmos DB アカウントの [OverView] ブレードを開いて [Gremlin Endpoint] の、URI をコピーします。

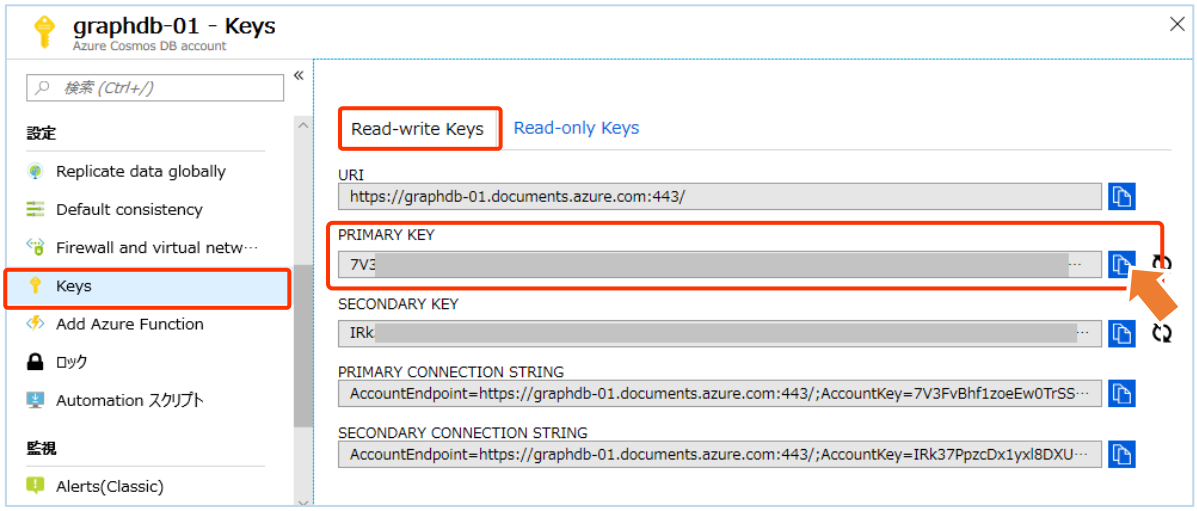


ワン ポイント
[Overview] ブレードには、[.NET SDK URI] と [Gremlin Endpoint] の 2 種類の URI が表示されますが、Gremlin.Net ドライバーで接続する場合、[Gremlin Endpoint] の URI を使用する必要があります。

4. Azure ポータルから コピーした URI を App.config ファイルの appSettings 要素 Endpoint の value 値として **[URI]** に貼り付けます。



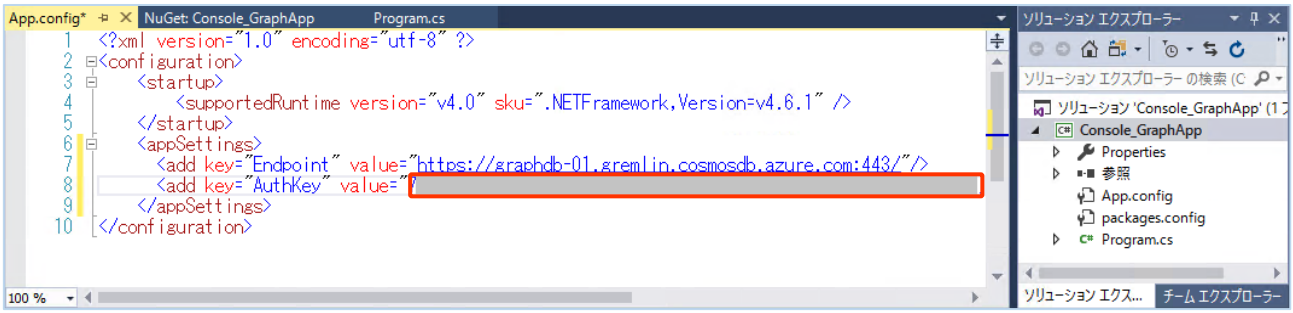
5. 再度 Azure ポータルに戻り、Azure Cosmos DB アカウントの [Keys] ブレードの [Read-write Keys] タブから [PRIMARY KEY] を取得します。



3.gremlin API SDK を使用する .NET コンソール アプリケーションの作成

ワンポイント
[Keys] ブレードには、プライマリとセカンダリの 2 種類のキーに対応した接続文字列が提供されています。これにより、管理者の変更などで、キーの再作成が必要になった場合、一時的にセカンダリに切り替え、プライマリ キーの再作成後、再び接続情報をプライマリに変更することで、アプリケーションからのデータ アクセスを中断しないで、キーを更新できるようにできます。 また、接続情報は、[Read-write Keys] タブと [Read-only Keys] タブで提供されています。[Read-only Keys] タブの接続文字列では、アカウントの読み取り操作のみが許可されます。データの追加、削除、置き換え操作が必要ないアプリケーションでは、読み取り専用の接続文字列を使用します。

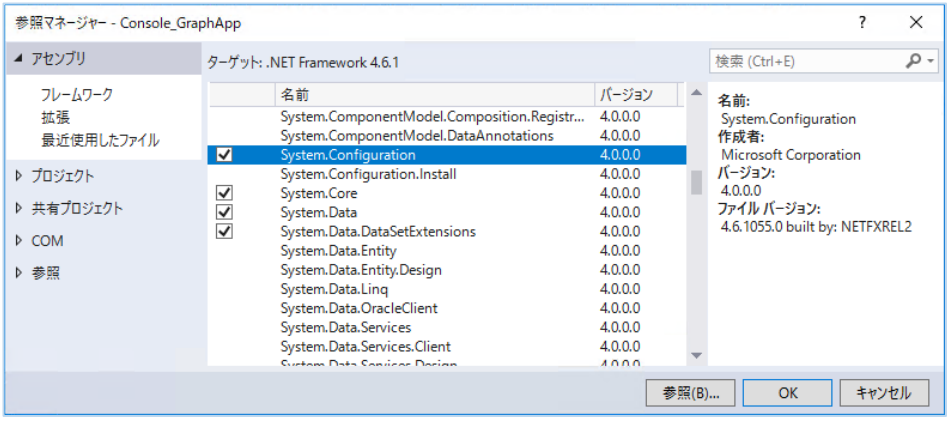
6. Azure ポータルから コピーした接続文字列を App.config ファイルの appSettings 要素 AuthKey の value 値として **[KEY]** に貼り付けます。



7. App.Config に記述した情報を変数に格納するため、Program.cs ファイルの Program クラスに次の静的メンバー変数を追加します。

```
private static string[] endpoint = ConfigurationManager.AppSettings["Endpoint"].Split(':');
private static string authKey = ConfigurationManager.AppSettings["AuthKey"];
```

ワン ポイント
App.Config に記述した Endpoint の値は、URL と ポート番号に分解して使用するため、String クラスの Split() メソッドを使用して「:」で分割して、配列に格納しています。 なお、この時、ConfigurationManager が認識されない場合は、ソリューションエクスプローラーのノードで [参照] を右クリックして、[参照の追加] で開く [参照マネージャー] から、「System.Configuration」を見つけて選択状態にしてください。



8. Endpoint の値を URL と ポート番号に分解して変数に格納するため、次の静的メンバー変数を追加します。

```
private static string hostname = endpoint[1].Trim('/');
private static int port = int.Parse(endpoint[2].Trim('/'));
```

9. 接続先のデータベース名とコレクション名を静的メンバー変数として追加します。

```
private static string database = "Adventureworks";
private static string collection = "Customers";
```

10. Vertex の数をカウントするテスト用のグラフ クエリを記述した Dictionary 型の静的メンバー変数を追加します。

```
private static Dictionary<string, string> testQueries = new Dictionary<string, string>
{
    { "Count Vertices",  "g.V().count()" },
};
```

11. 2 章で Customers グラフを作成している Azure Cosmos DB アカウントを Gremlin サーバーとしてインスタンス化するために、Main() メソッドに次のコードを追加します。

```
// Customers グラフを作成した Azure Cosmos DB アカウントを Gremlin サーバーとしてインスタンス化する
var gremlinServer = new GremlinServer(hostname, port, enableSsl: true,
                                      username: "/dbs/" + database + "/colls/" + collection,
                                      password: authKey);
```

ワン ポイント

接続する Azure CosmosDB アカウントに Adventureworks データベースと Customers グラフが存在しない場合、自習書の「2.2.1 Azure ポータルからの Azure Cosmos DB データベース アカウントの作成」と「2.2.2 Azure ポータルからのグラフ データベースとコンテナーの作成」を実行してください。

12. 実行するグラフ クエリを記述した Dictionary 型のデータを gremlinQueries にセットするために、Main() メソッドに次のコードを追加します。

```
// 実行するクエリを記述した Dictionary 型のデータを gremlinQueries にセット
Dictionary<string, string> gremlinQueries = testQueries;
```

13. Gremlin サーバーから Gremlin クライアントを作成するために、Main() メソッドに次のコードを追加します。

```
// Gremlin サーバーから Gremlin クライアントを作成する
using (var gremlinClient = new GremlinClient(gremlinServer, new GraphSON2Reader(),
                                new GraphSON2Writer(), GremlinClient.GraphSON2MimeType))
{
}
```

ワン ポイント

この後、using ステートメントで作成した Gremlin クライアントを使用してグラフ クエリを実行するようにコードを仕上げていきます。

14. using ステートメントで作成した Gremlin クライアントのメソッドを使用してグラフ クエリを実行するために、using ブロック内に、次のコードを追加します。

```
// gremlinQueries から、実行するクエリを取得する
foreach (var query in gremlinQueries)
{
    // 実行するクエリを表示する
    Console.WriteLine(String.Format("Running this query: {0}: {1}", query.Key, query.Value));

    // クエリを実行し、結果を取得する
    var resultSet = gremlinClient.SubmitAsync<dynamic>(query.Value).Result;

    // 結果セットの表示
    if (resultSet.Count > 0)
    {
        Console.WriteLine("¥tResult:");
        foreach (var result in resultSet)
        {
            // ネストした Dictionary 型として得られた結果を JSON 形式に変換して表示する
            string output = JsonConvert.SerializeObject(result);
            Console.WriteLine($"¥t{output}");
        }
        Console.WriteLine();
    }
}
```

15. コードの終了時にコマンド プロンプトが閉じないようにするため、Main() メソッドの using ブロックの後に、次のコードを追加します。

```
// プログラム終了時のメッセージ出力
Console.WriteLine("Done. Press any key to exit...");
Console.ReadLine();
```

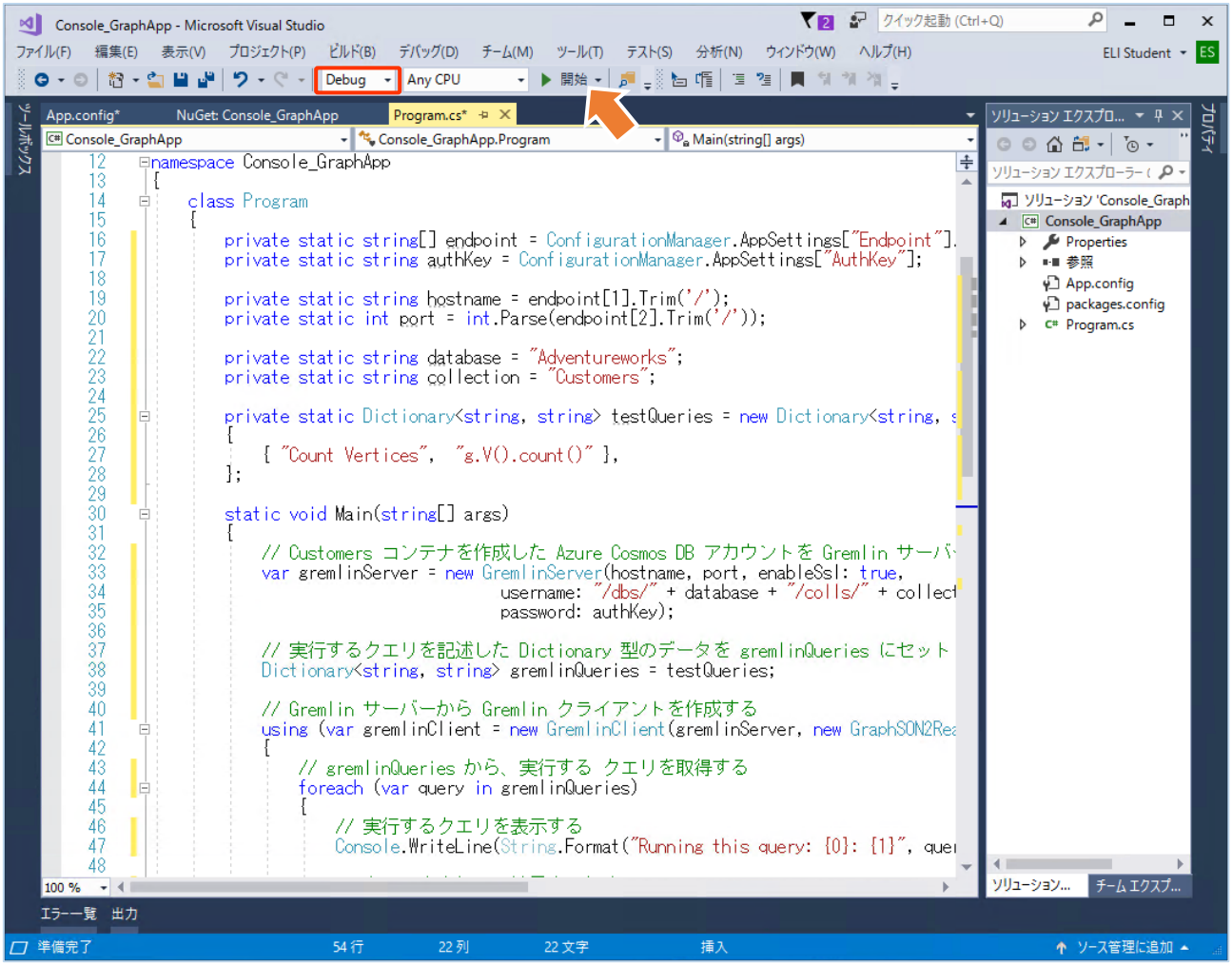
16. ここまでの作業で記述されたコードは以下のようになります。

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using System.Configuration;

using Gremlin.Net.Driver;
using Gremlin.Net.Driver.Exceptions;
using Gremlin.Net.Structure.IO.GraphSON;
using Newtonsoft.Json;

namespace Console_GraphApp
{
    class Program
```

```
{
    private static string[] endpoint = ConfigurationManager.AppSettings["Endpoint"].Split(':');
    private static string authKey = ConfigurationManager.AppSettings["AuthKey"];

    private static string hostname = endpoint[1].Trim('/');
    private static int port = int.Parse(endpoint[2].Trim('/'));

    private static string database = "Adventureworks";
    private static string collection = "Customers";

    private static Dictionary<string, string> testQueries = new Dictionary<string, string>
    {
        { "Count Vertices",  "g.V().count()" },
    };

    static void Main(string[] args)
    {
        // Customers グラフを作成した Azure Cosmos DB アカウントを Gremlin サーバーとしてインスタンス化する
        var gremlinServer = new GremlinServer(hostname, port, enableSsl: true,
                                              username: "/dbs/" + database + "/colls/" + collection,
                                              password: authKey);

        // 実行するクエリを記述した Dictionary 型のデータを gremlinQueries にセット
        Dictionary<string, string> gremlinQueries = testQueries;

        // Gremlin サーバーから Gremlin クライアントを作成する
        using (var gremlinClient = new GremlinClient(gremlinServer, new GraphSON2Reader(),
                                    new GraphSON2Writer(), GremlinClient.GraphSON2MimeType))
        {
            // gremlinQueries から、実行するクエリを取得する
            foreach (var query in gremlinQueries)
            {
                // 実行するクエリを表示する
                Console.WriteLine(String.Format("Running this query: {0}: {1}", query.Key, query.Value));

                // クエリを実行し、結果を取得する
                var resultSet = gremlinClient.SubmitAsync<dynamic>(query.Value).Result;

                // 結果セットの表示
                if (resultSet.Count > 0)
                {
                    Console.WriteLine("¥tResult:");
                    foreach (var result in resultSet)
                    {
                        // ネストした Dictionary 型として得られた結果を JSON 形式に変換して表示する
                        string output = JsonConvert.SerializeObject(result);
```

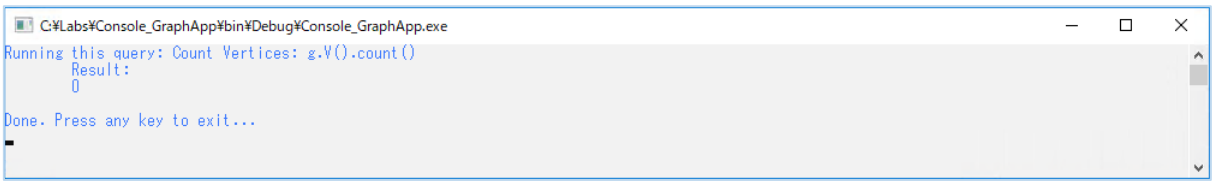
```
                    Console.WriteLine($"¥t{output}");
                }
                Console.WriteLine();
            }
        }
    }
    // プログラム終了時のメッセージ出力
    Console.WriteLine("Done. Press any key to exit...");
    Console.ReadLine();
  }
 }
}
```

17. 記述したコードを検証するために、[ソリューション構成] が「Debug」に設定されていることを確認して、[開始] ボタンをクリックしてデバッグ実行します。



18. コマンド プロンプトが起動して、次のメッセージが表示されたら、キーボードの任意のキーをクリックして、コンソール アプリケーションのデバッグ実行を止めます。

3.gremlin API SDK を使用する .NET コンソール アプリケーションの作成



ワン ポイント
2 章の「2.2.5.3 変更と削除の操作」で、V().drop() ステップを使用して、Vertex、Edge、プロパティの削除を実行している場合、Result に 0 が返されます。以降の手順で、2 章で作成した Vertex と同じものをアプリケーション コードで作成しますので、ここで、グラフに Vertex が残っている場合は、「V().drop()」を実行してクリーンアップしておいてください。 また、デバッグ実行時に「System.Net.WebSockets.WebSocketException: 'サーバーが状態コード '101' を返すはずが、状態コード '200' を返しました。'」というメッセージが表示された場合、App.config ファイルの appSettings 要素 Endpoint の value 値に記述された URI を確認してください。データベースとコンテナーを作成しているアカウントの「Gremlin Endpoint」URI を記述する必要があります。 なお、クエリの結果セット (ResultSet) は、クエリを実行した Gremlin サーバーから返された結果を表しています。ResultSet には、列挙可能なデータと Dictionary 型のステータス属性が含まれています。以降の手順では、ResultSet の StatusAttributes プロパティに含まれるステータス属性を表示するコードを追加します。

19. ResultSet のステータス属性から、ステータス属性、ステータス コード、および、クエリ実行で消費された要求ユニット (RU) の数を表示する PrintStatusAttributes メソッドを作成するために、Program クラス (Main メソッドの外) に、次のコードを追加します。

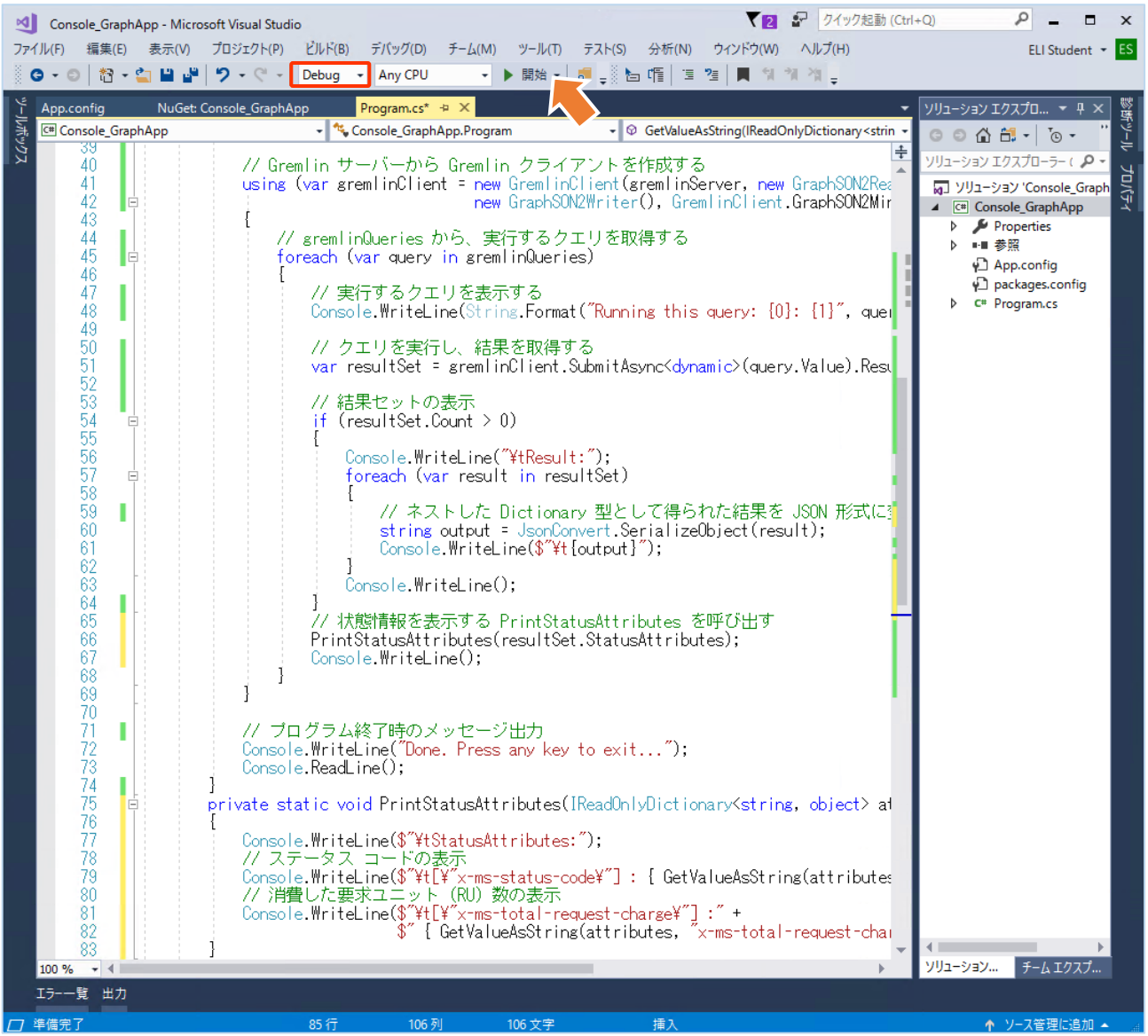
```
private static void PrintStatusAttributes(IReadOnlyDictionary<string, object> attributes)
{
    Console.WriteLine($"¥tStatusAttributes:");
    // ステータス コードの表示
    Console.WriteLine($"¥t[¥"x-ms-status-code¥"] : { GetValueAsString(attributes, "x-ms-status-code")}");
    // 消費した要求ユニット (RU) 数の表示
    Console.WriteLine($"¥t[¥"x-ms-total-request-charge¥"] :" +
                      $" { GetValueAsString(attributes, "x-ms-total-request-charge")}");
}

public static string GetValueAsString(IReadOnlyDictionary<string, object> dictionary, string key)
{
    // Dictionary 型を JSON に変換する
    return JsonConvert.SerializeObject(GetValueOrDefault(dictionary, key));
}

// Dictionary 型の値を確認し、データがある場合は、Dictionary 型を返し、データがない場合は NULL 値を返す
public static object GetValueOrDefault(IReadOnlyDictionary<string, object> dictionary, string key)
{
    if (dictionary.ContainsKey(key))
    {
        return dictionary[key];
    }
    return null;
}
```

3.gremlin API SDK を使用する .NET コンソール アプリケーションの作成

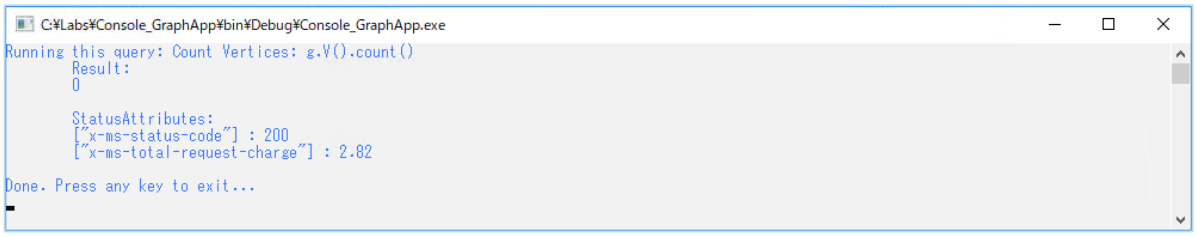
20. Gremlin クライアントを作成した using ブロック内の結果セットを表示する if ブロック後に PrintStatusAttributes メソッドを呼び出すための次のコードを追加します。

```
// 状態情報を表示する PrintStatusAttributes を呼び出す
PrintStatusAttributes(resultSet.StatusAttributes);
Console.WriteLine();
```

21. 記述したコードを検証するために、[ソリューション構成] が「Debug」に設定されていることを確認して、[開始] ボタンをクリックしてデバッグ実行します。



22. コマンド プロンプトが起動して、次のメッセージが表示されたら、キーボードの任意のキーをクリックして、コンソール アプリケーションのデバッグ実行を止めます。



ワン ポイント

x-ms-status-code は、ステータスコードで、x-ms-total-request-charge は、クエリが消費した要求ユニット (RU) 数を表しています。

3.4 アプリケーション コードによる Vertex と Edge の追加

ここでは、前の手順で作成したアプリケーション コードを使用し、グラフに Vertex と Edge を追加するグラフ クエリを実行します。

1. Program クラスのメンバー変数が宣言されている箇所に、グラフに Vertex を追加するためのグラフ クエリを記述した Dictionary 型の静的メンバー変数を追加します。

```
private static Dictionary<string, string> addVertices = new Dictionary<string, string>
{
    { "Cleanup",          "g.V().drop()" },
    { "Add Customer 1",
        "g.addV('customer').property('id', '1').property('name', 'Orlando Gee').property('age', 28)" },
    { "Add Customer 2",
        "g.addV('customer').property('id', '2').property('name', 'Keith Harris').property('age', 27)" },
    { "Add Customer 3",
        "g.addV('customer').property('id', '3').property('name', 'Donna Carreras').property('age', 32)" },
    { "Add Customer 4",
        "g.addV('customer').property('id', '4').property('name', 'Janet Gates').property('age', 37)" },
    { "Add Customer 5",
        "g.addV('customer').property('id', '5').property('name', 'Lucy Harrington').property('age', 32)" },
    { "Add Customer 6",
        "g.addV('customer').property('id', '6').property('name', 'Rosmarie Carroll').property('age', 27)" },
    { "Add Product 1",
        "g.addV('product').property('id', 'p-780').property('name', 'Mountain-200').property('price', 2319.99)" },
    { "Add Product 2",
        "g.addV('product').property('id', 'p-867').property('name', 'Mountain Shorts').property('price', 69.99)" },
};
```

ワン ポイント

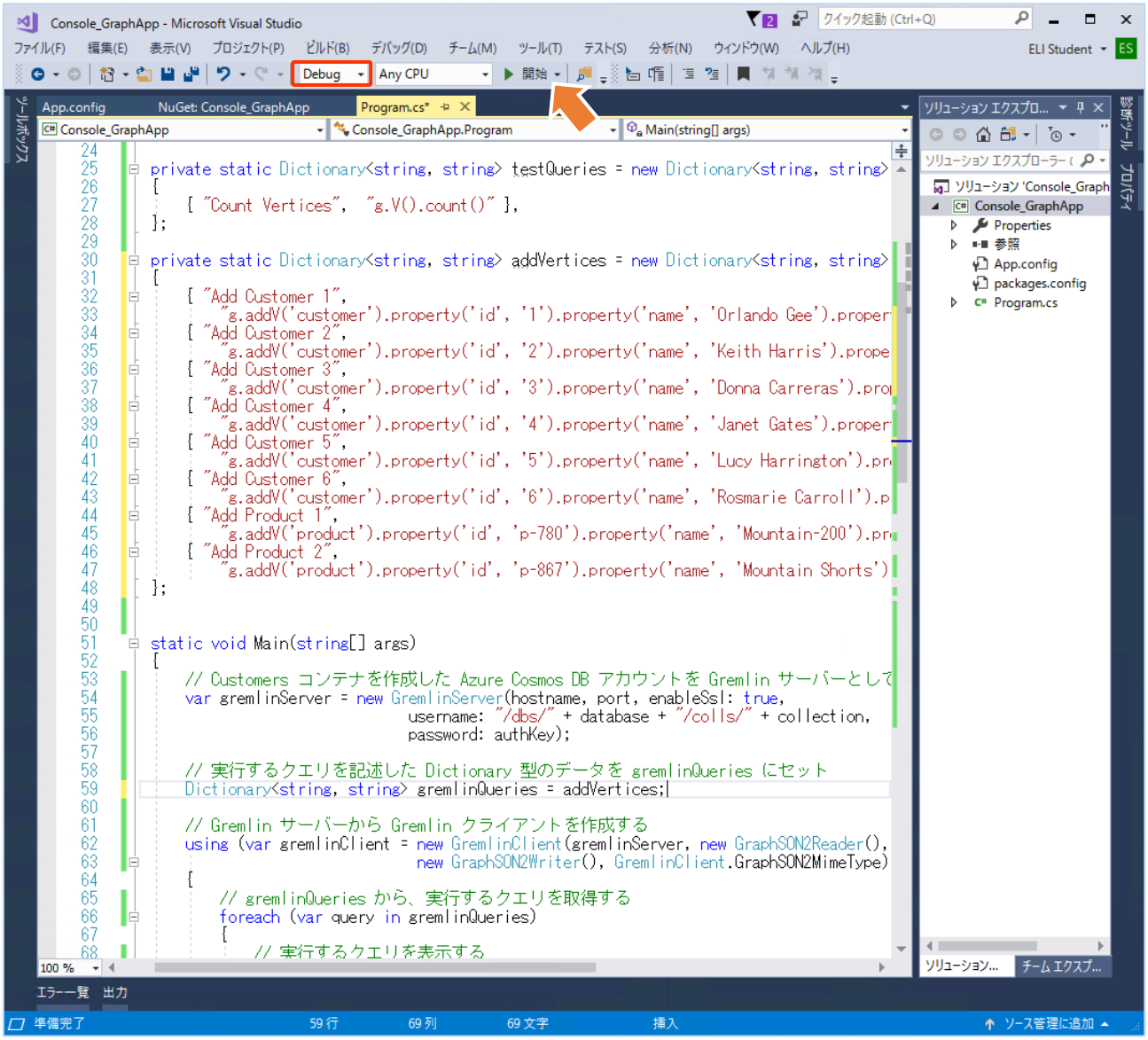
8 個の addV() ステップにより、customer ラベルの Vertex が 6 個と product ラベルの Vertex が 2 個追加されます。
なお、このコードを自習書からコピー ペーストした場合、インデントがずれてしまう可能性があります。複数行をまとめてインデントする場合、対象の複数行を選択状態にして、Tab キーをクリックしてください。

2. グラフに Vertex を追加するグラフ クエリを記述した Dictionary 型の静的メンバー変数 「**addVertices**」 を gremlinQueries にセットするために、「実行するクエリを記述した Dictionary 型のデータを gremlinQueries にセット」と書かれたコメントの後のコードを以下の内容に変更します。

```
// 実行するクエリを記述した Dictionary 型のデータを gremlinQueries にセット
Dictionary<string, string> gremlinQueries = addVertices;
```

3.gremlin API SDK を使用する .NET コンソール アプリケーションの作成

3. 記述したコードを検証するために、[ソリューション構成] が「Debug」に設定されていることを確認して、[開始] ボタンをクリックしてデバッグ実行します。

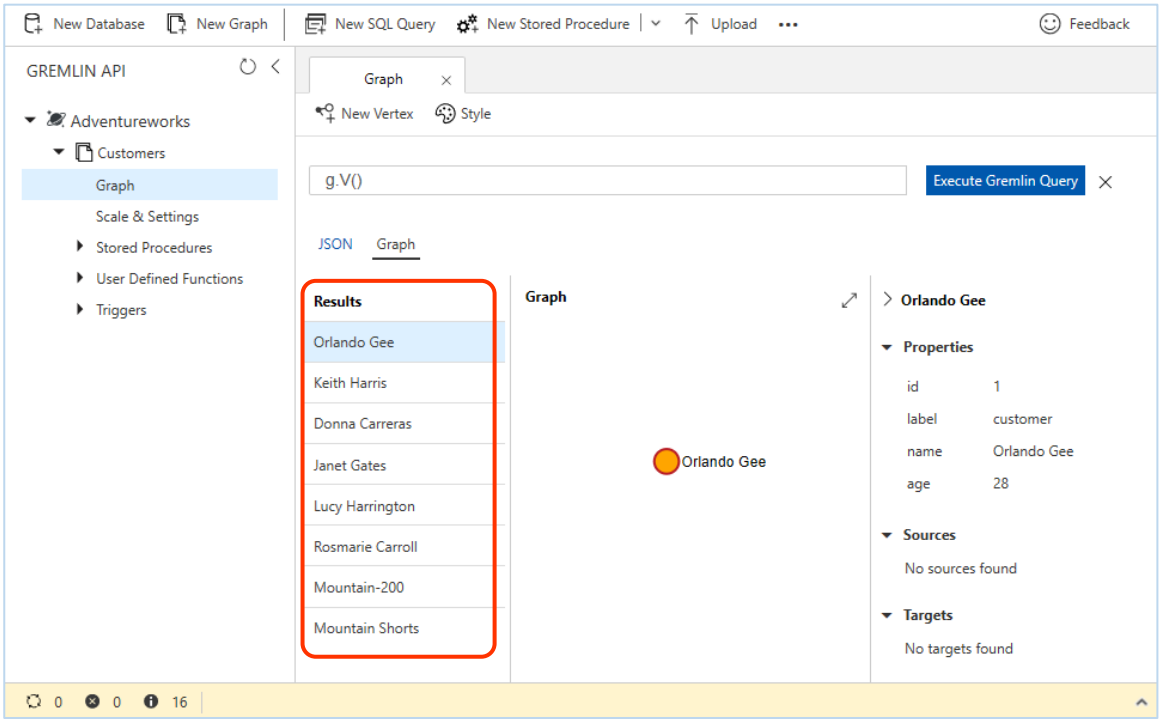


4. コマンド プロンプトが起動して、次のメッセージが表示されたら、キーボードの任意のキーをクリックして、コンソール アプリケーションのデバッグ実行を止めます。

3.gremlin API SDK を使用する .NET コンソール アプリケーションの作成

```
Running this query: Add Customer 4: g.addV('customer').property('id', '4').property('name', 'Janet Gates').property('age', 37)
        Result:
        {"id":"4","label":"customer","type":"vertex","properties":{"name":[{"id":"778cf79f-cdf9-401e-a51c-c5925c6acacc","value":"Janet Gates"}],"a
ge":[{"id":"a9810077-5812-4b9b-abb4-3e27e41243c6","value":37}]}}

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 10.379999999999999

Running this query: Add Customer 5: g.addV('customer').property('id', '5').property('name', 'Lucy Harrington').property('age', 32)
        Result:
        {"id":"5","label":"customer","type":"vertex","properties":{"name":[{"id":"ecfcd29d-3434-4e0b-b815-5579fa30a948","value":"Lucy Harrington"}
],"age":[{"id":"6b89b360-0bd0-4d70-938d-7736113a6592","value":32}]}}

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 10.379999999999999

Running this query: Add Customer 6: g.addV('customer').property('id', '6').property('name', 'Rosmarie Carroll').property('age', 27)
        Result:
        {"id":"6","label":"customer","type":"vertex","properties":{"name":[{"id":"886fbc45-c27d-4865-b506-345f8f7c8fd2","value":"Rosmarie Carroll"
}],"age":[{"id":"0cad5874-7fd3-4cc0-b458-7d4674d7600b","value":27}]}}

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 10.379999999999999

Running this query: Add Product 1: g.addV('product').property('id', 'p-780').property('name', 'Mountain-200').property('price', 2319.99)
        Result:
        {"id":"p-780","label":"product","type":"vertex","properties":{"name":[{"id":"6b83fc1b-3e93-45b8-9722-21eb64abf9dd","value":"Mountain-200"}
],"price":[{"id":"b680bc2b-0f3a-49b9-94c4-e95fa076c453","value":2319.99}]}}

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 10.379999999999999

Running this query: Add Product 2: g.addV('product').property('id', 'p-867').property('name', 'Mountain Shorts').property('price', 69.99)
        Result:
        {"id":"p-867","label":"product","type":"vertex","properties":{"name":[{"id":"6c3e9ecc-5608-44fa-9abb-f30ca40eb915","value":"Mountain Short
s"}],"price":[{"id":"8edd9f0d-53ad-4e5a-96a2-f86386a17f7e","value":69.99}]}}

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 10.379999999999999

Done. Press any key to exit...
```

5. アプリケーション コードの実行結果を確認するために、[Data Explorer] ブレードで、「**g.V()**」と入力して、[Execute Gremlin Query] をクリックしてグラフ クエリを実行します。



ワン ポイント
グラフに追加された 6 個の customer ラベルの Vertex と 2 個の product ラベルの Vertex の name プロパティが [Results] 領域に表示されます。

6. Program クラスのメンバー変数が宣言されている箇所に、グラフに Edge を追加するためのグラフ クエリを記述した Dictionary 型の静的メンバー変数を追加します。

```
private static Dictionary<string, string> addEdges = new Dictionary<string, string>
{
    { "Add Follows 1",
        "g.V().has('name', 'Orlando Gee').addE('follows').to(g.V().has('name', 'Keith Harris'))" },
    { "Add Follows 2",
        "g.V().has('name', 'Keith Harris').addE('follows').to(g.V().has('name', 'Orlando Gee'))" },
    { "Add Follows 3",
        "g.V().has('name', 'Donna Carreras').addE('follows').to(g.V().has('name', 'Keith Harris'))" },
    { "Add Follows 4",
        "g.V().has('name', 'Janet Gates').addE('follows').to(g.V().has('name', 'Rosmarie Carroll'))" },
    { "Add Follows 5",
        "g.V().has('name', 'Lucy Harrington').addE('follows').to(g.V().has('name', 'Keith Harris'))" },
    { "Add Follows 6",
        "g.V().has('name', 'Rosmarie Carroll').addE('follows').to(g.V().has('name', 'Orlando Gee'))" },
    { "Add Bought 1",  "g.V('1').addE('bought').to(g.V('p-780')).property('rating', 4)" },
    { "Add Bought 2",  "g.V('2').addE('bought').to(g.V('p-780')).property('rating', 5)" },
    { "Add Bought 3",  "g.V('5').addE('bought').to(g.V('p-780')).property('rating', 4)" },
    { "Add Bought 4",  "g.V('2').addE('bought').to(g.V('p-867')).property('rating', 3)" },
    { "Add Bought 5",  "g.V('5').addE('bought').to(g.V('p-867')).property('rating', 5)" },
};
```

ワン ポイント
11 個の addE() ステップにより、follows ラベルの Edge が 6 個と bought ラベルの Vertex が 5 個追加されます。

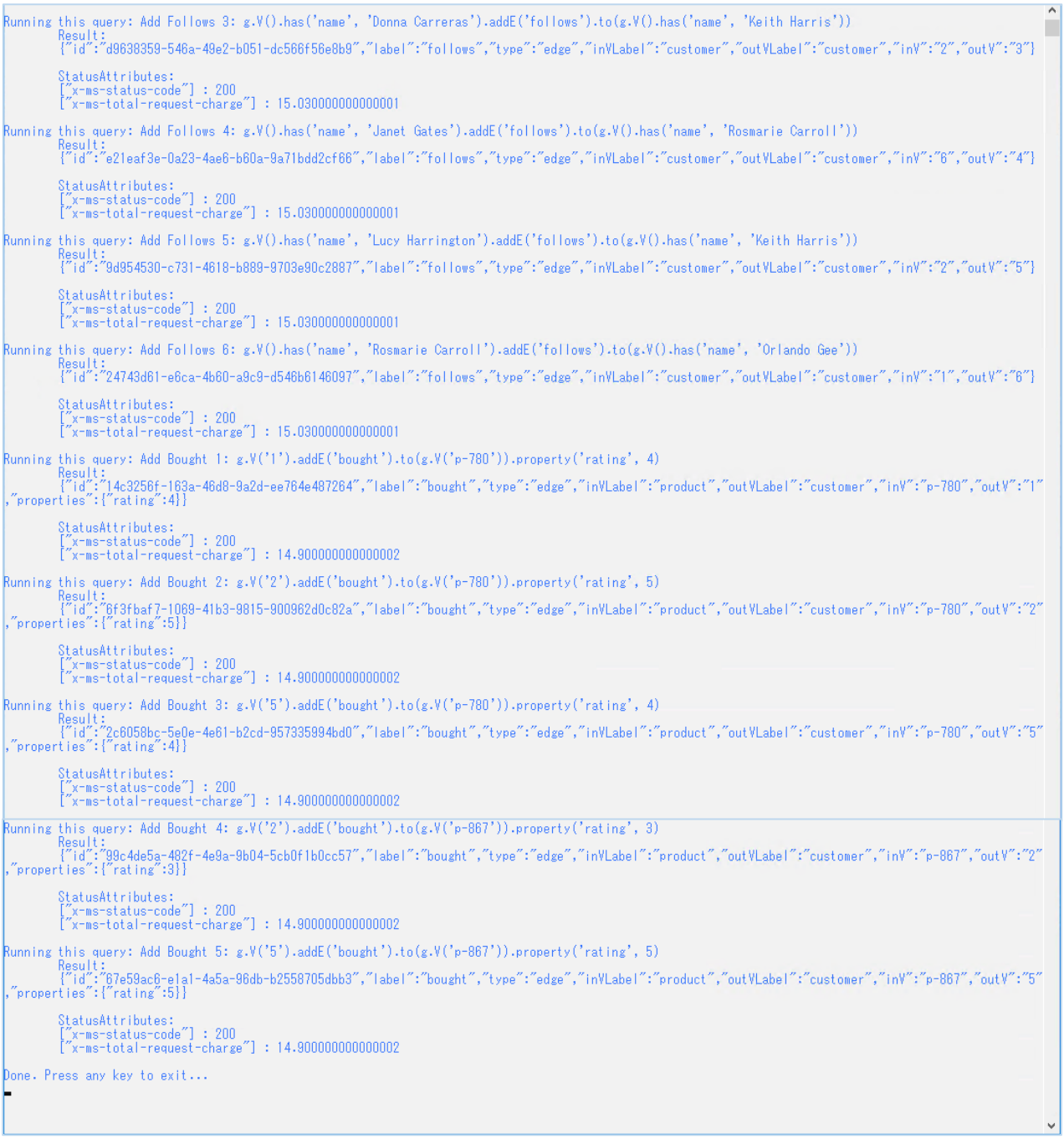
7. グラフに Vertex を追加するグラフ クエリを記述した Dictionary 型の静的メンバー変数 「**addEdges**」 を gremlinQueries にセットするために、「実行するクエリを記述した Dictionary 型のデータを gremlinQueries にセット」と書かれたコメントの後のコードを以下の内容に変更します。

```
// 実行するクエリを記述した Dictionary 型のデータを gremlinQueries にセット
Dictionary<string, string> gremlinQueries = addEdges;
```

8. 記述したコードを検証するために、[ソリューション構成] が「Debug」に設定されていることを確認して、[開始] ボタンをクリックしてデバッグ実行します。

9. コマンド プロンプトが起動して、次のメッセージが表示されたら、キーボードの任意のキーをクリックして、コンソール アプリケーションのデバッグ実行を止めます。

3.gremlin API SDK を使用する .NET コンソール アプリケーションの作成



3.5 アプリケーション コードによるトラバーサル

ここでは、引き続き、ここまでの作業で追加した Vertex と Edge を使用して、アプリケーション コードからグラフ クエリを実行します。

1. Program クラスのメンバー変数が宣言されている箇所に、単純なグラフ クエリを記述した Dictionary 型の静的メンバー変数を追加します。

```
private static Dictionary<string, string> simpleQueries = new Dictionary<string, string>
{
      { "Count Vertices", "g.V().count()" },
      { "Count Edges",    "g.E().count()" },
      { "Filter",         "g.V().hasLabel('customer').has('age', lt(30)).values('name')" },
      { "Grouping",       "g.V().group().by(label).by('name')" },
      { "Enumeration",    "g.V().has('name',within('Donna Carreras', 'Lucy Harrington', 'Janet Gates'))" +
                          ".values('age')" },
      { "Aggregate",      "g.V().has('name',within('Donna Carreras', 'Lucy Harrington', 'Janet Gates'))" +
                          ".values('age').mean()" },
};
```

2. グラフに Vertex を追加するグラフ クエリを記述した Dictionary 型の静的メンバー変数 「**simpleQueries**」 を gremlinQueries にセットするために、「実行するクエリを記述した Dictionary 型のデータを gremlinQueries にセット」と書かれたコメントの後のコードを以下の内容に変更します。

```
// 実行するクエリを記述した Dictionary 型のデータを gremlinQueries にセット
Dictionary<string, string> gremlinQueries = simpleQueries;
```

3. 記述したコードを検証するために、[ソリューション構成] が「Debug」に設定されていることを確認して、[開始] ボタンをクリックしてデバッグ実行します。
4. コマンド プロンプトが起動して、次のメッセージが表示されたら、キーボードの任意のキーをクリックして、コンソール アプリケーションのデバッグ実行を止めます。

```
Running this query: Enumeration: g.V().has('name',within('Donna Carreras', 'Lucy Harrington', 'Janet Gates')).values('age')
        Result:
        32
        37
        32

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 5.73

Running this query: Aggregate: g.V().has('name',within('Donna Carreras', 'Lucy Harrington', 'Janet Gates')).values('age').mean()
        Result:
        33.666666666666664

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 5.92

Done. Press any key to exit...
```

5. Program クラスのメンバー変数が宣言されている箇所に、トラバーサルを実行するグラフ クエリを記述した Dictionary 型の静的メンバー変数を追加します。

```
private static Dictionary<string, string> traversalQueries = new Dictionary<string, string>
{
        { "Traversal 1",   "g.V().has('name','Orlando Gee').outE('follows').inV().values('name')" },
        { "Traversal 2",   "g.V().has('name','Orlando Gee').out('follows').values('name')" },
        { "Traversal 3",   "g.V().has('name','Orlando Gee').in('follows').values('name')" },
        { "Traversal 4",   "g.V().has('name','Orlando Gee').in('follows').in('follows').values('name')" },
        { "Traversal 5",   "g.V().has('name','Orlando Gee').as('self').in('follows').in('follows')" +
                           ".where(neq('self')).values('name')" },
        { "Traversal 6",   "g.V().has('name', 'Orlando Gee').out('bought').values('name')" },
        { "Traversal 7",   "g.V().has('name', 'Orlando Gee').as('self').out('bought').in('bought')" +
                           ".where(neq('self')).values('name')" },
        { "Traversal 8",   "g.V().has('name', 'Orlando Gee').as('self').out('bought').in('bought')" +
                           ".where(neq('self')).out('bought').dedup().values('name')" },
};
```

6. グラフに Vertex を追加するグラフ クエリを記述した Dictionary 型の静的メンバー変数「**traversalQueries**」を gremlinQueries にセットするために、「実行するクエリを記述した Dictionary 型のデータを gremlinQueries にセット」と書かれたコメントの後のコードを以下の内容に変更します。

```
// 実行するクエリを記述した Dictionary 型のデータを gremlinQueries にセット
Dictionary<string, string> gremlinQueries = traversalQueries;
```

7. 記述したコードを検証するために、[ソリューション構成] が「Debug」に設定されていることを確認して、[開始] ボタンをクリックしてデバッグ実行します。
8. コマンド プロンプトが起動して、次のメッセージが表示されたら、キーボードの任意のキーをクリックして、コンソール アプリケーションのデバッグ実行を止めます。

```
C:¥Labs¥Console_GraphApp¥bin¥Debug¥Console_GraphApp.exe
Running this query: Traversal 1: g.V().has('name','Orlando Gee').outE('follows').inV().values('name')
        Result:
        "Keith Harris"

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 11.08
```

3.gremlin API SDK を使用する .NET コンソール アプリケーションの作成

```
Running this query: Traversal 2: g.V().has('name','Orlando Gee').out('follows').values('name')
        Result:
        "Keith Harris"

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 13.64

Running this query: Traversal 3: g.V().has('name','Orlando Gee').in('follows').values('name')
        Result:
        "Keith Harris"
        "Rosmarie Carroll"

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 13.93

Running this query: Traversal 4: g.V().has('name','Orlando Gee').in('follows').in('follows').values('name')
        Result:
        "Orlando Gee"
        "Donna Carreras"
        "Lucy Harrington"
        "Janet Gates"

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 24.14

Running this query: Traversal 5: g.V().has('name','Orlando Gee').as('self').in('follows').in('follows').where(neq('self')).values('name')
        Result:
        "Donna Carreras"
        "Lucy Harrington"
        "Janet Gates"

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 25.240000000000002

Running this query: Traversal 6: g.V().has('name', 'Orlando Gee').out('bought').values('name')
        Result:
        "Mountain-200"

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 13.64

Running this query: Traversal 7: g.V().has('name', 'Orlando Gee').as('self').out('bought').in('bought').where(neq('self')).values('name')
        Result:
        "Keith Harris"
        "Lucy Harrington"

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 24.46

Running this query: Traversal 8: g.V().has('name', 'Orlando Gee').as('self').out('bought').in('bought').where(neq('self')).out('bought').dedup().values(
'name')
        Result:
        "Mountain-200"
        "Mountain Shorts"

        StatusAttributes:
        ["x-ms-status-code"] : 200
        ["x-ms-total-request-charge"] : 35.02

Done. Press any key to exit...
```

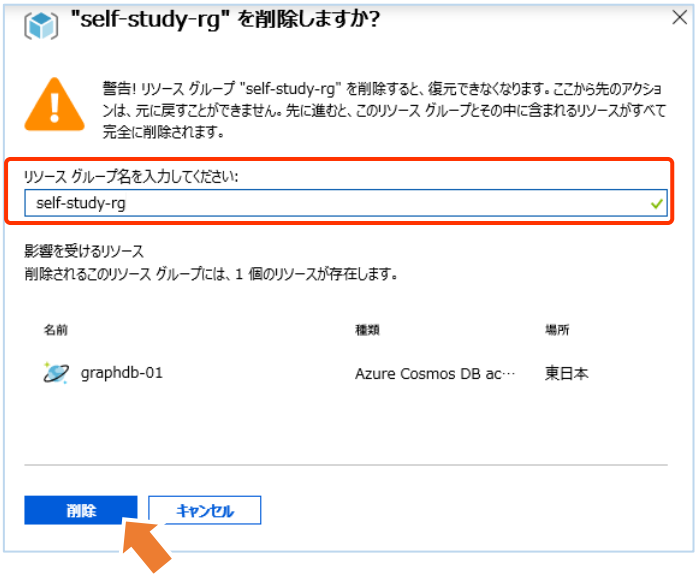
3.6 リソースの削除手順

自習書の手順に従って作成したリソースを残しておくと課金が継続することに注意してください。最後の手順では、作成したリソース グループを削除します。個々のリソースを個別に削除することもできますが、リソース グループを削除することで、そのリソース グループ内のすべてのリソースをまとめて削除できます。なお、検証を続けたい場合は、検証の完了後に行ってください。

1. Azure ポータルのハブ メニューから [リソース グループ] をクリックして、表示されるリソース グループのリストから自習書用に作成したリソース グループをクリックします。

2. 自習書用に作成したリソース グループのブレードで [リソース グループの削除] をクリックします。

3. 次のメッセージが表示されます。ここで [リソース グループ名を入力してください] に削除するリソース グループ名を入力して、[削除] ボタンをクリックします。



ワン ポイント

自習書用に作成したリソース グループを削除すると、その中にあるすべてのリソースが削除され、元に戻すことはできないことに注意してください。

参考文献

本書の執筆で参考にした文献、サイトは以下のとおりです。自習書の復習と今後の学習の際に参考にしてください。

Azure Cosmos DB のドキュメント

Azure Cosmos DB の機能全般を説明しているドキュメントです。このサイトで提供される様々な API とプログラミング モデルに対応するチュートリアルとサンプルは、Azure Cosmos DB を使用したソリューションの構築に役立つでしょう。

TinkerPop3 Documentation

Gremlin のグラフ データベースの構造とグラフ クエリ言語を解説している TinkerPop3 のサイトです。

Getting started with Azure Cosmos DB: Graph API

Gremlin.Net ドライバーを使用して Azure Cosmos DB Graph API アカウントに接続し、Vertex と Edge の作成、読み取り、更新、削除のための基本的なグラフ クエリを実行する GitHub に公開されたサンプル コードです。