

한국 마이크로소프트

Microsoft Technical Trainer

Enterprise Skills Initiative

Microsoft Azure

Azure Cosmos DB

이 문서는 Microsoft Technical Trainer팀에서 ESI 교육 참석자분들에게 제공해 드리는 문서입니다.

요약

이 내용들은 표시된 날짜에 Microsoft에서 검토된 내용을 바탕으로 하고 있습니다. 따라서, 표기된 날짜 이후에 시장의 요구사항에 따라 달라질 수 있습니다. 이 문서는 고객에 대한 표기된 날짜 이후에 변화가 없다는 것을 보증하지 않습니다.

이 문서는 정보 제공을 목적으로 하며 어떠한 보증을 하지는 않습니다.

저작권에 관련된 법률을 준수하는 것은 고객의 역할이며, 이 문서를 마이크로소프트의 사전 동의 없이 어떤 형태(전자 문서, 물리적인 형태 막론하고) 어떠한 목적으로 재 생산, 저장 및 다시 전달하는 것은 허용되지 않습니다.

마이크로소프트는 이 문서에 들어있는 특허권, 상표, 저작권, 지적 재산권을 가집니다. 문서를 통해 명시적으로 허가된 경우가 아니면, 어떠한 경우에도 특허권, 상표, 저작권 및 지적 재산권은 다른 사용자에게 허여되지 않습니다.

© 2022 Microsoft Corporation All right reserved.

Microsoft®는 미합중국 및 여러 나라에 등록된 상표입니다.

이 문서에 기재된 실제 회사 이름 및 제품 이름은 각 소유자의 상표일 수 있습니다.

문서 작성 연혁

날짜	버전	작성자	변경 내용
2022.02.15	0.3.0	우진환	TASK 01 작성
2022.02.16	0.7.0	우진환	TASK 02 작성
2022.02.18	1.0.0	우진환	TASK 03 작성
2022.04.03	1.1.0	우진환	TASK 04 작성, Azure 포털 UI 변경 적용

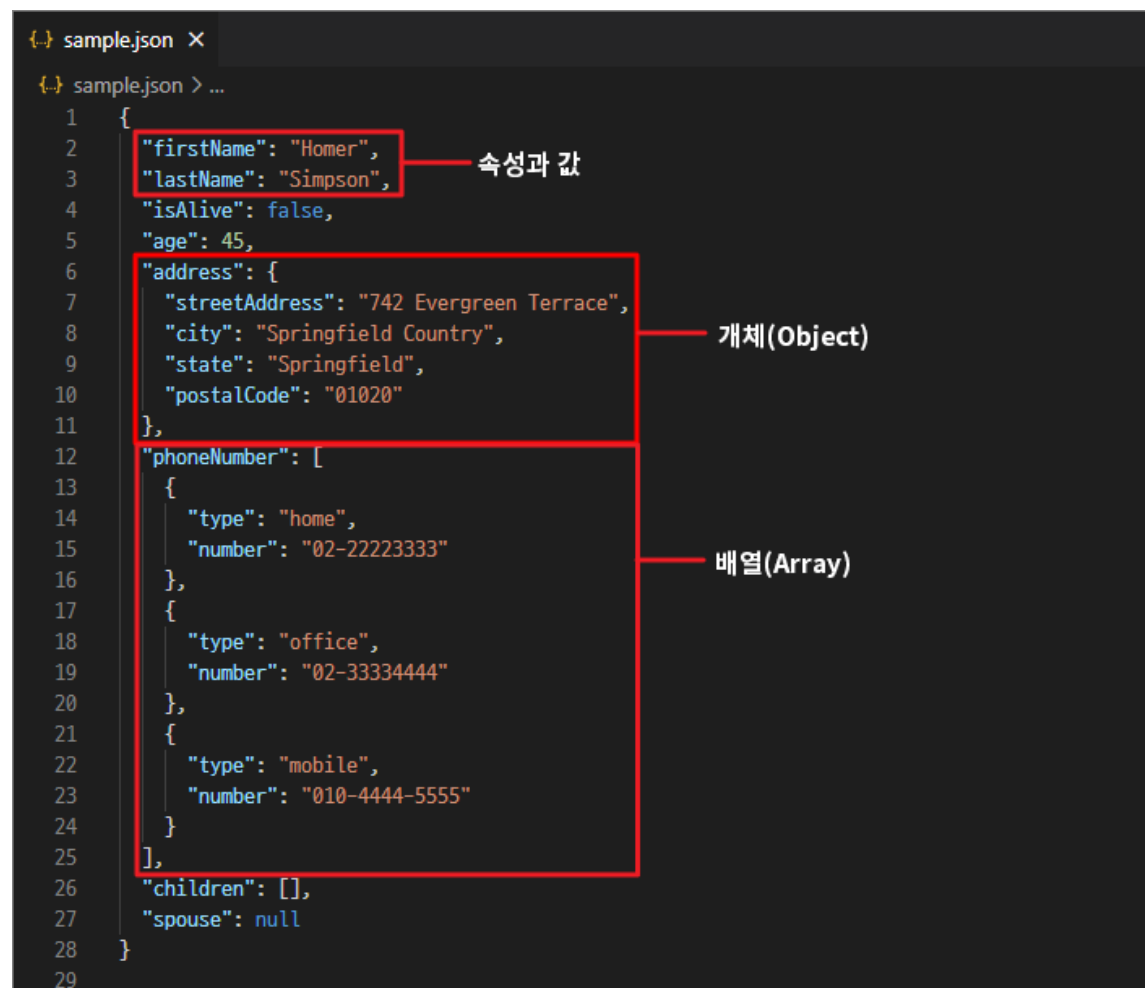
목차

AZURE COSMOS DB 개요.....	5
AZURE COSMOS DB 리소스 모델	6
요청 단위(REQUEST UNITS).....	8
파티션(PARTITION).....	9
CHANGE FEED	11
TASK 01. AZURE COSMOS DB 기본 작업.....	12
TASK 02. VISUAL STUDIO CODE에서 COSMOS DB 개발	21
TASK 03. COSMOS DB 쿼리.....	28
TASK 04. 리소스 정리.....	35

Azure Cosmos DB 개요

Azure Cosmos DB는 턴키 방식의 글로벌 배포 및 투명한 다중 마스터 복제(multi-master replication)가 포함된 완전히 관리되는 데이터베이스 서비스입니다. Cosmos DB는 대규모, 글로벌 복제, 짧은 대기 시간의 데이터베이스 용도로 디자인되었으며, 데이터베이스 내의 트랜잭션 데이터에 대해 운영 및 분석 워크로드와 AI를 실행할 수 있습니다.

Cosmos DB는 Document Database입니다. Document DB는 JSON 형식의 문서로 데이터를 저장 및 쿼리하도록 디자인된 비관계형 데이터베이스 유형입니다. **Document**는 JSON 형식이며, JSON은 사람이 읽을 수 있는 텍스트를 사용하여 속성-값(attribute-value) 쌍 및 배열(array) 데이터 유형으로 구성된 데이터 개체를 전송하는 개방형 표준 파일 형식입니다. JSON 형식에서 **object**나 **array**를 사용하여 필요에 맞는 작업을 수행할 수 있습니다. 따라서 현재 모든 API를 프로그래밍하기 위해서 가장 일반적으로 사용되는 문서 형식입니다.



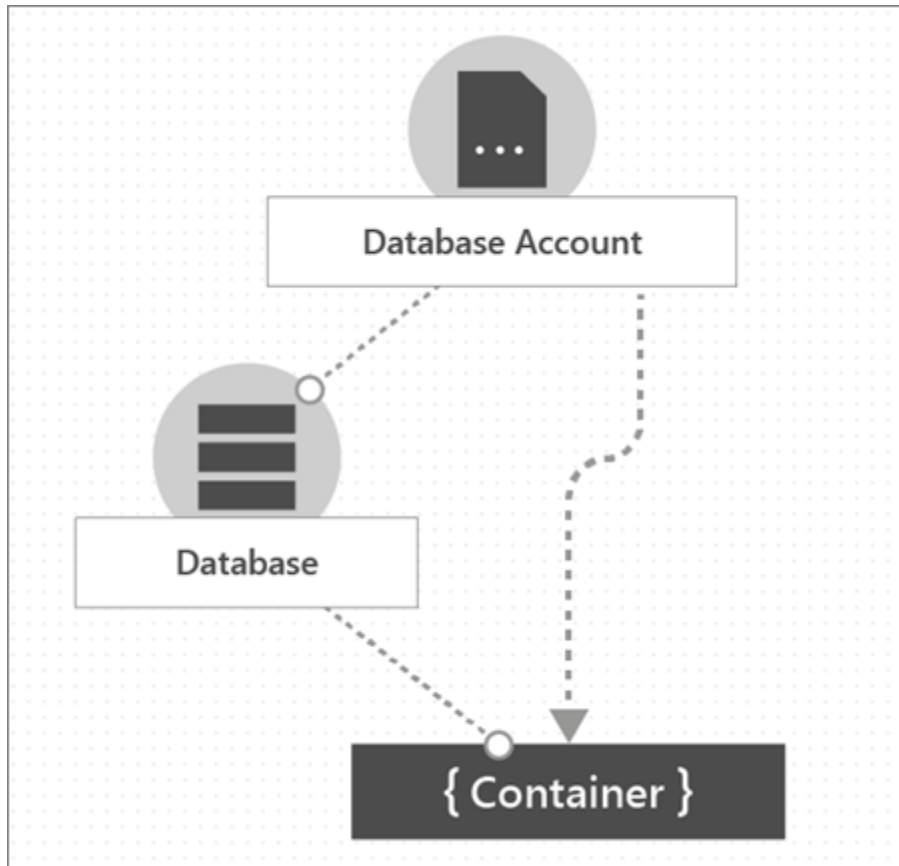
Azure Cosmos DB의 주요 기능은 아래와 같습니다.

- 턴키 글로벌 분산: 데이터베이스는 Azure의 모든 주요 지역에 투명하게 분산됩니다. 따라서 복제에 대해 걱정할 필요가 없으며 몇 번의 클릭으로 자동으로 분산을 구성할 수 있습니다.
- 지역 사용 가능성: 데이터베이스를 모든 Azure 지역에 글로벌로 분산하고 사용할 수 있습니다.

- Always On: 전 세계에 걸쳐 99.999%의 읽기 및 쓰기 가용성을 보장합니다.
- 탄력적 규모: 기본적으로 초당 수천에서 수억 요청까지 확장할 수 있습니다.
- 낮은 대기 시간 보장: 99번째 백분위 수에서 10ms 미만의 읽기 및 쓰기를 보장합니다.
- 일관성 옵션: 성능과 일관적인 복제 간의 균형을 선택할 수 있습니다. Azure Cosmos DB 복제 프로토콜은 5가지의 일관성 모델(**Strong**, **Bounded staleness**, **Session**, **Consistent Prefix**, **Eventual**)을 제공합니다. 각 모델에는 일관성과 성능 간의 교환(tradeoff)이 있습니다. **Strong** 모델은 가장 강력한 일관성을 제공하고 **Eventual**은 가장 높은 가용성과 낮은 대기 시간, 처리량을 제공합니다.
- 스키마나 인덱스 관리 없음: 스키마 불가지론(schema-agnostic)으로 디자인되었습니다. 즉 관리할 스키마나 인덱스가 없습니다. 따라서 일반적인 테이블 정의 열 정의, 열의 유형이 없으며 **Document**에 대해 작업하는 스키마가 없기 때문에 각 **Document**는 다른 속성을 가질 수 있습니다. 또한 모든 데이터를 자동으로 인덱싱합니다. 인덱싱 정책으로 이를 재정의할 수 있지만 기본적으로는 데이터 인덱싱에 대해 걱정할 필요가 없습니다.
- 다중 API: Cosmos DB의 가장 중요한 기능 중 하나는 여러 API를 지원한다는 것입니다. Cosmos DB를 만들 때 데이터 작업에 사용할 API의 종류를 결정합니다. 기본 핵심 API는 SQL이며 Cassandra, MongoDB, Graph를 사용하는 경우 Gremlin을 사용할 수 있습니다. 또한 Azure Table 스토리지를 더 좋은 성능으로 마이그레이션하는 경우 Azure Table Storage API를 사용할 수 있습니다.

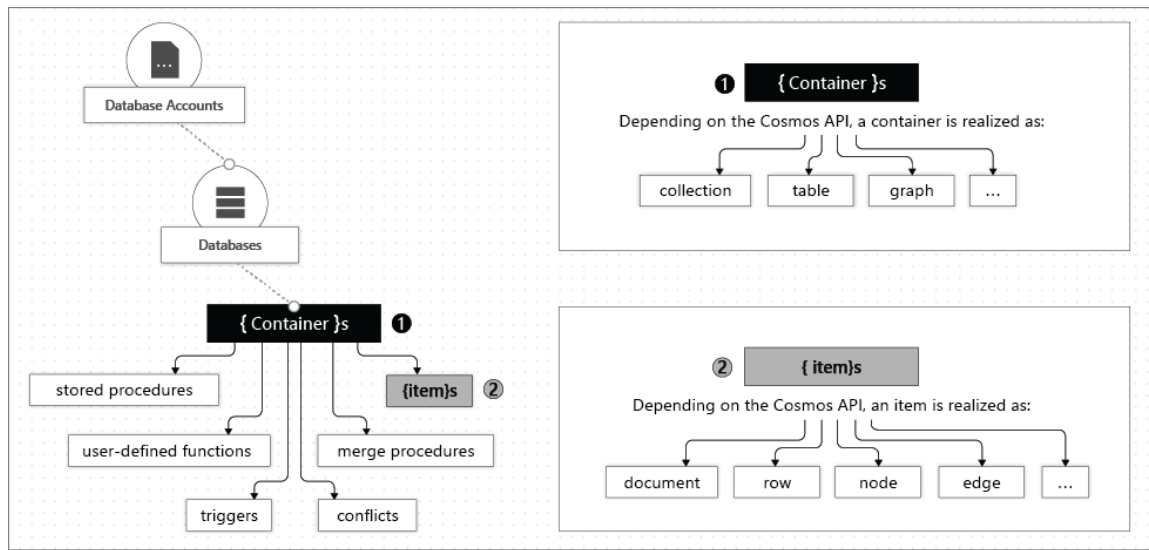
Azure Cosmos DB 리소스 모델

Azure Cosmos DB 컨테이너는 확장성의 기본 단위입니다. 컨테이너에서 사실상 무제한의 프로비저닝된 처리량(RU)과 스토리지를 가질 수 있습니다. Azure Cosmos DB는 프로비저닝된 처리량 및 스토리지를 탄력적으로 확장하기 위해 지정한 논리적 파티션 키(logical partition key)를 사용하여 컨테이너를 투명하게 분할합니다. 현재 Azure 구독에서 최대 50개의 Azure Cosmos DB 계정을 만들 수 있지만 이는 소프트웨어 제한이기 때문에 지원 요청을 통해 늘릴 수 있습니다. 단일 Azure Cosmos DB 계정은 사실상 무제한의 데이터 및 프로비저닝된 처리량을 관리할 수 있습니다. 데이터 및 프로비저닝된 처리량을 관리하기 위해 계정 아래에 하나 이상의 Azure Cosmos 데이터베이스를 만들고 해당 데이터베이스 내에 하나 이상의 컨테이너를 만들 수 있습니다.



Azure Cosmos DB의 구조는 매우 단순합니다.

- 데이터베이스 계정은 최상위 리소스이며 데이터베이스 계정 아래에 하나 이상의 데이터베이스가 있을 수 있습니다. 이를 SQL 데이터베이스와 동일하게 생각할 수 있습니다.
- 컨테이너(container): 데이터베이스 아래에는 컨테이너가 있으며 이를 테이블이라고 생각할 수 있습니다.
- 항목(item): 각 테이블에는 항목(item)이라는 데이터 행(row)이 있습니다. 앞서 JSON 형식으로 되어 있는 Document를 하나의 항목으로 볼 수 있습니다. 또한 저장 프로시저, 사용자 정의 함수, 트리거, 병합 프로시저 등을 사용할 수 있습니다.
- 이 구조에서 중요한 것은 API에 따라 컨테이너에 포함되는 것이 다르다는 것입니다. 예를 들어 Gremlin을 API로 선택하면 포함되는 내용은 graph이며, SQL을 선택하면 컬렉션이 포함되고, Table을 선택하면 table 스토리지가 포함됩니다.
- 따라서 선택한 API에 따라 항목(item) 컬렉션은 document, 행(row), 노드(node), 에지(edge)가 됩니다.

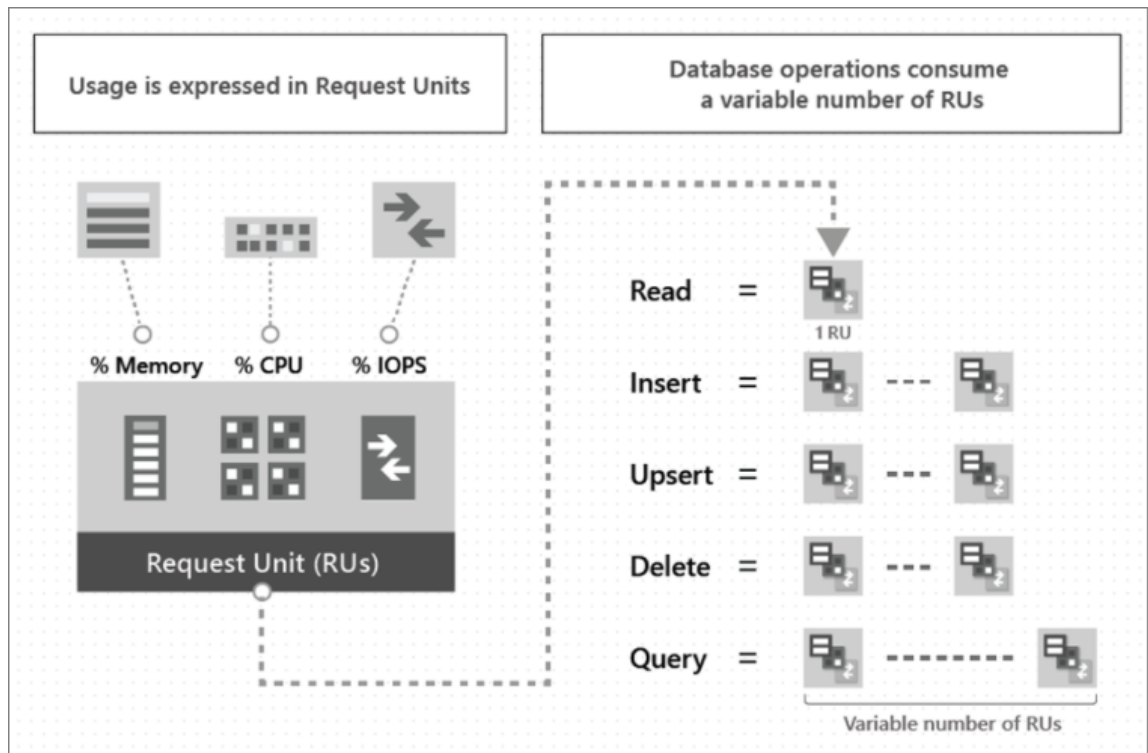


요청 단위(Request Units)

모든 데이터베이스 작업은 Azure Cosmos DB에서 정규화되며 요청 단위(RUs; Request Units)로 표시됩니다.

RU는 Azure Cosmos DB에서 지원하는 데이터베이스 작업을 수행하는데 필요한 CPU, IOPS, 메모리와 같은 시스템 리소스를 추상화하는 성능 통화(currency)입니다. Azure Cosmos DB는 다른 Azure 서비스와 달리 코어 수, 메모리 크기를 선택하지 않기 때문에 처리량 측정값인 RU만 선택하면 됩니다.

1KB 항목(item)에 대한 포인트 읽기(예를 들어 ID와 파티션 키 값으로 단일 항목 가져오기)를 수행하는 비용이 1 RU입니다. 모든 다른 데이터베이스 작업은 RU를 사용하여 비슷한 방법으로 비용이 할당됩니다. Azure Cosmos DB 컨테이너와 상호작용하는데 사용하는 API와 관계없이 비용은 항상 RU로 측정됩니다. 또한 데이터베이스 작업이 읽기, 포인트 읽기, 쿼리에 관계없이 비용은 항상 RU로 측정됩니다.

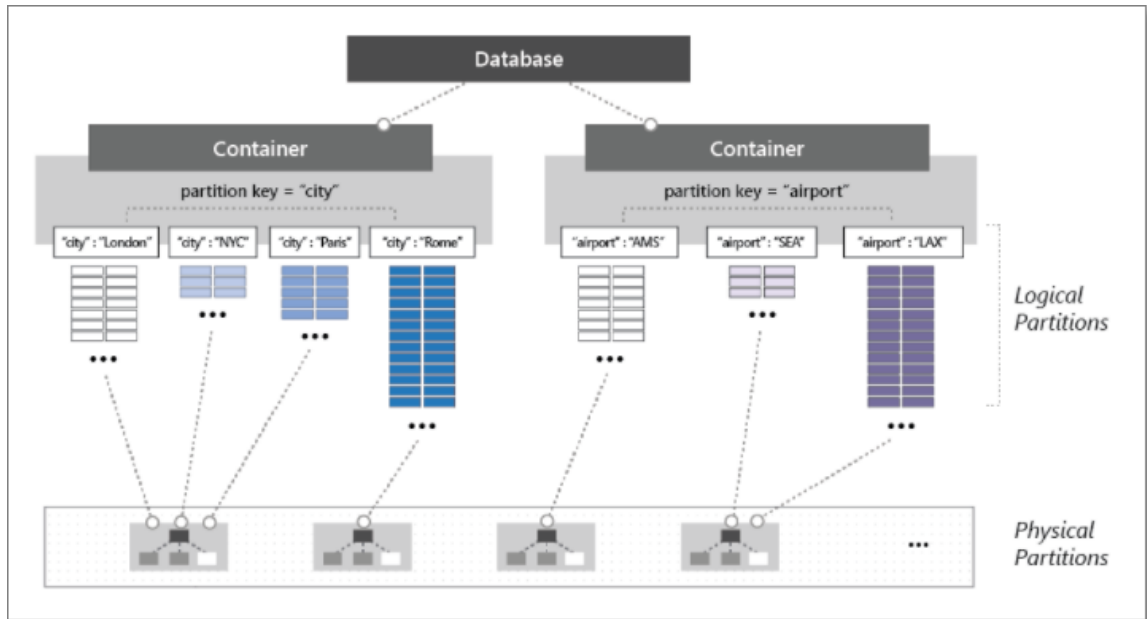


파티션(Partition)

Azure Cosmos DB는 애플리케이션의 성능 요구 사항을 충족하기 위해 데이터베이스 내의 개별 컨테이너를 확장하기 위한 방법으로 파티셔닝(partitioning)을 사용합니다. 파티셔닝(partitioning)에서 컨테이너의 항목은 논리적 파티션(logical partition)이라는 개별적인 하위 집합으로 나뉩습니다. 논리적 파티션(logical partition)은 컨테이너의 각 항목(item)에 할당된 파티션 키(partition key)의 값을 기반으로 형성됩니다. 논리적 파티션(logical partition)의 모든 항목(item)은 동일한 파티션 키(partition key) 값을 가집니다.

예를 들어 컨테이너에 항목(item)이 있고 각 항목(item)에는 **UserID** 속성에 대한 고유한 값이 있는 경우, 컨테이너의 항목(item)에서 **UserID**는 파티션 키(partition key)로 사용되며 1000개의 고유한 **UserID** 값이 있는 경우 컨테이너에 1000개의 고유한 논리적 파티션(logical partition)이 생성됩니다.

항목(item)의 논리적 파티션(logical partition)을 결정하는 파티션 키(partition key) 외에도 컨테이너의 각 항목(item)에는 항목 ID (item ID)가 있으며 항목 ID (item ID)는 논리적 파티션 내에서 고유합니다. 파티션 키(partition key)와 항목 ID (item ID)를 결합하면 항목(item)을 고유하게 식별하는 항목(item)의 인덱스가 생성됩니다. 즉 파티션 키(partition key)를 선택하는 것은 애플리케이션의 성능에 영향을 미치는 중요한 결정 사항입니다.



논리 파티션(logical partition)은 동일한 파티션 키(partition key)를 가지고 있는 항목(item)의 집합으로 구성됩니다. 예를 들어 식품 영양에 대한 데이터가 포함된 컨테이너의 모든 항목에 **foodGroup** 속성이 포함되어 있는 경우 **foodGroup**을 컨테이너의 파티션 키로 사용할 수 있습니다. 이 경우 소고기 제품, 구운 제품, 소시지, 런치미트와 같이 **foodGroup**에 대한 특정 값을 가지고 있는 항목의 그룹은 개별적인 논리 파티션(logical partition)을 형성합니다.

논리 파티션(logical partition)은 데이터베이스 트랜잭션의 범위도 정의합니다. 스냅샷 격리가 있는 트랜잭션을 사용하여 논리 파티션 내의 항목을 업데이트할 수 있습니다. 새 항목이 컨테이너에 추가되면 새 논리 파티션이 시스템에 의해 투명하게 생성됩니다. 또한 기본 데이터가 삭제될 때 논리 파티션 삭제에 대해 사용자가 신경 쓸 필요는 없습니다.

컨테이너의 논리 파티션 수에는 제한이 없으며 각 논리 파티션은 최대 20GB의 데이터를 저장할 수 있습니다. 좋은 파티션 키 선택은 가능한 값의 범위가 넓습니다. 예를 들어 모든 항목에 **foodGroup** 속성이 포함된 컨테이너에서 소고기 제품이라는 논리 파티션의 데이터는 최대 20GB까지 증가할 수 있습니다. 가능한 값의 범위가 넓은 파티션 키를 선택하면 컨테이너를 확장할 수 있습니다.

컨테이너는 물리 파티션(physical partition)에 데이터와 처리량을 분산하여 확장됩니다. 내부적으로 하나 이상의 논리 파티션(logical partition)이 단일 물리 파티션에 매핑됩니다. 일반적으로 더 많은 논리 파티션을 가진 작은 컨테이너는 단일 물리 파티션만 필요합니다. 논리 파티션과 달리 물리 파티션은 시스템의 내부 구현이며 Azure Cosmos DB에 의해 완전히 관리됩니다.

컨테이너의 물리 파티션 수는 다음에 따라 다릅니다

- 프로비저닝된 처리량 수(각 개별 물리 파티션은 초당 최대 10,000 RU를 처리량을 제공할 수 있음). 물리 파티션의 10,000 RU 제한은 논리 파티션에도 10,000 RU 제한이 있음을 의미합니다. 이는 각 논리 파티션이 하나의 물리 파티션에만 매핑되기 때문입니다.
- 총 데이터 스토리지(각 개별 물리 파티션은 50GB 데이터를 저장할 수 있음)

물리 파티션은 Azure Cosmos DB에 의해 완전히 관리되어 솔루션을 개발할 때 물리 파티션을 제어할 수 기

때문에 물리 파티션에 대해 고려할 것은 없습니다. 대신 파티션 키를 고려해야 합니다. 처리량 소비(consumption)가 논리 파티션 전체에 고르게 분산되는 파티션 키를 선택하면 물리 파티션 전체의 처리량 소비가 균형을 이루도록 할 수 있습니다.

컨테이너에서 물리 파티션의 수는 제한이 없습니다. 프로비저닝된 처리량 혹은 데이터 크기가 증가하면 Azure Cosmos DB는 기존 물리 파티션을 분할하여 새 물리 파티션을 자동으로 만듭니다. 물리 파티션 분할은 애플리케이션의 가용성에 영향을 미치지 않습니다. 물리 파티션이 분할된 후 단일 논리 파티션 내의 모든 데이터는 여전히 동일한 물리 파티션에 저장됩니다. 물리 파티션 분할은 단순히 논리 파티션과 물리 파티션에 대한 새 매핑을 생성합니다.

컨테이너에 프로비저닝된 처리량은 물리 파티션 간에 균등하게 분할됩니다. 요청을 고르게 분산하지 않는 파티션 키 디자인은 너무 많은 요청을 "hot"이 되는 파티션의 하위 집합으로 보낼 수 있습니다. "hot" 파티션은 프로비저닝된 처리량의 비효율적인 사용으로 이어져 속도 제한 및 더 높은 비용을 발생할 수 있습니다.

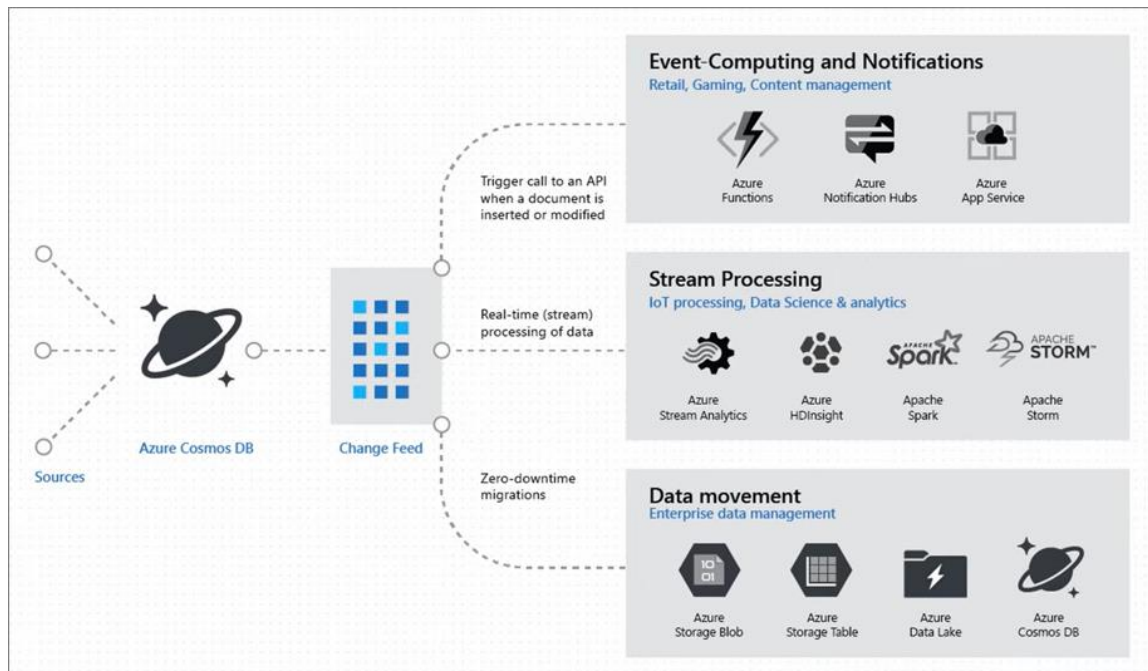
각 물리 파티션은 복제본의 집합인 **replica set**으로 구성됩니다. 각 **replica set**은 데이터베이스 엔진의 인스턴스를 호스팅합니다. **replica set**은 물리 파티션 내에 저장된 데이터를 내구성 있고 고가용성을 제공하며 일관성을 보장해 주도록 합니다. 물리 파티션을 구성하는 각 **replica**는 파티션의 스토리지 할당량을 상속합니다. 물리 파티션의 모든 **replica**는 물리 파티션에 할당된 처리량을 집합적으로 지원하며 Azure Cosmos DB는 자동으로 **replica set**을 관리합니다. 일반적으로 더 작은 컨테이너는 단일 물리 파티션만 필요하지만 여전히 최소 4개의 **replica**가 있습니다.

Change feed

Azure Cosmos DB의 change feed는 컨테이너에 대한 변경 사항이 발생한 순서대로 기록되는 지속적인 레코드입니다. Azure Cosmos DB의 change feed는 모든 변경 사항에 대해 Azure Cosmos 컨테이너를 수신 대기(listening)하는 방식으로 작동합니다. 그런 다음 수정된 순서대로 변경된 **document**의 정렬된 목록을 출력합니다. 지속된 변경 사항은 비동기식 및 증분식으로 처리될 수 있으며 출력은 병렬 처리를 위해 하나 이상의 소비자(consumer)에게 배포될 수 있습니다. change feed는 다음과 같은 Azure Cosmos DB API와 클라이언트 SDK에서 지원됩니다.

클라이언트 드라이버	SQL API	Azure Cosmos DB's API for Cassandra	Azure Cosmos DB's API for MongoDB	Gremlin API	Table API
.NET	예	예	예	예	아니요
Java	예	예	예	예	아니요
Python	예	예	예	예	아니요
Node/JS	예	예	예	예	아니요

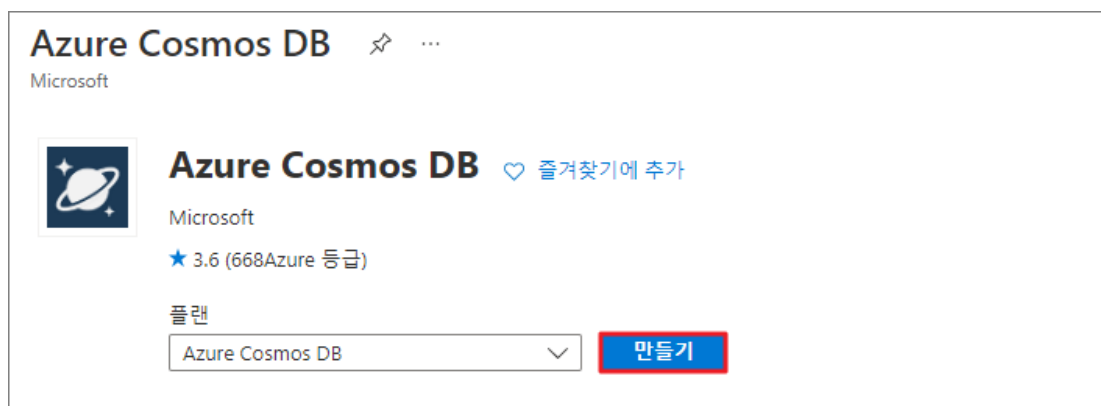
change feed는 Azure Functions, Stream Analytics, HDInsights, Databricks와 같은 Azure 리소스에 쉽게 연결할 수 있게 해 줍니다. 기본적으로 Cosmos DB의 변경 사항에 대해 반응하고 계산을 수행하고 분석에 대한 결과를 저장할 수 있습니다.



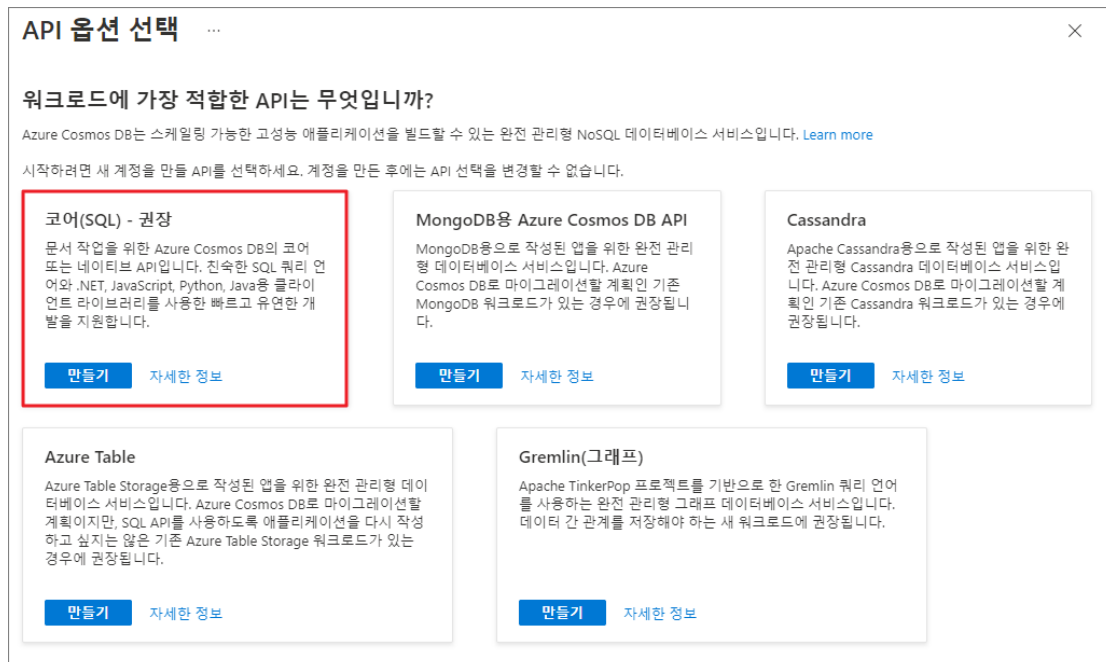
TASK 01. Azure Cosmos DB 기본 작업

이 작업에서는 Azure Cosmos DB를 만들고 테스트를 위해 항목(item)을 추가한 후 간단한 쿼리를 수행하여 사용되는 RU에 대해 평가합니다.

1. Azure 포털에서 [리소스 만들기]를 클릭하고 "Cosmos DB"를 검색합니다. [Azure Cosmos DB] 블레이드에서 [만들기]를 클릭합니다.



2. [API 옵션 선택] 블레이드에서 [코어(SQL) - 권장] 타일의 [만들기]를 클릭합니다.



3. [Azure Cosmos DB 계정 만들기 - 코어(SQL)] 블레이드의 [기본 사항] 탭에서 아래와 같이 구성한 후 [다음]을 클릭합니다.

- [프로젝트 세부 정보 - 리소스 그룹]: "새로 만들기"를 클릭한 후 "04_cosmosDbRg"를 입력합니다.
- [인스턴스 세부 정보 - 계정 이름]: 중복되지 않는 고유한 이름을 입력합니다. 계정 이름은 데이터베이스에 대한 공용 URL이 되기 때문에 신중하게 결정해야 합니다.
- [인스턴스 세부 정보 - 위치]: (Asia Pacific) Korea Central
- [인스턴스 세부 정보 - 용량 모드]: 프로비저닝된 처리량
- [인스턴스 세부 정보 - 무료 계층 할인 적용]: 적용 안 함
- [인스턴스 세부 정보 - 총 계층 처리량 제한]: 사용하지 않음

Azure Cosmos DB 계정 만들기 - 코어(SQL)

기본 사항 전역 배포 네트워크 백업 정책 암호화 태그 검토 + 만들기

Azure Cosmos DB는 확장 가능한 고성능 애플리케이션을 구축하기 위한 완전 관리형 NoSQL 데이터베이스 서비스입니다. 무제한으로 경신할 수 있는 30일 평가판을 사용 [해 보세요](#). 여러 컨테이너가 포함되어 있고 데이터베이스당 월별 \$24부터 시작하는 프로덕션 환경을 이용하세요. [자세한 정보](#)

프로젝트 세부 정보

배포된 리소스 및 비용을 관리할 구독을 선택합니다. 폴더와 같은 리소스 그룹을 사용하여 모든 리소스를 구성하고 관리합니다.

구독 * Azure Pass - 스폰서십

리소스 그룹 * (신규) 04_cosmosDbRg
[새로 만들기](#)

인스턴스 세부 정보

계정 이름 * kormttcosmos

위치 * (Asia Pacific) Korea Central

용량 모드 ☒ 프로비저닝된 처리량 ☐ Serverless
[용량 모드에 대한 자세한 정보](#)

With Azure Cosmos DB free tier, you will get the first 1000 RU/s and 25 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated \$64/month discount per account.

무료 계층 할인 적용 ☐ 적용 ☒ 적용 안 함

총 계정 처리량 제한 ☐ 이 계정에서 프로비저닝할 수 있는 총 처리량 제한

i 이 제한은 프로비저닝된 처리량과 관련된 예상치 못한 요금이 발생하지 않도록 합니다. 계정을 만든 후 이 한도를 업데이트하거나 제거할 수 있습니다.

4. [전역 배포] 옵션에서 기본 설정을 유지하고 [다음]을 클릭합니다. 이 설정은 Cosmos DB를 생성한 후 나중에 다시 검토합니다.

Azure Cosmos DB 계정 만들기 - 코어(SQL)

기본 사항 전역 배포 네트워크 백업 정책 암호화 태그 검토 + 만들기

전역 배포

계정의 전역 배포 및 국가별 설정을 구성합니다. 계정을 만든 후에 이 설정을 변경할 수도 있습니다.

지역 중복 ☐ 사용 ☒ 사용 안 함

다중 지역 쓰기 ☐ 사용 ☒ 사용 안 함

가용성 영역 ☐ 사용 ☒ 사용 안 함

5. [네트워크] 탭에서 "모든 네트워크"를 선택하고 [다음]을 클릭합니다.

Azure Cosmos DB 계정 만들기 - 코어(SQL)

기본 사항 전역 배포 네트워크 백업 정책 암호화 태그 검토 + 만들기

네트워크 연결

공용 IP 주소 또는 서비스 엔드포인트를 통해 공개적으로 또는 프라이빗 엔드포인트를 사용하여 비공개적으로 Cosmos DB 계정에 연결할 수 있습니다.

연결 방법 * ☒ 모든 네트워크
☐ 퍼블릭 엔드포인트(선택한 네트워크)
☐ 프라이빗 엔드포인트

모든 네트워크에서 이 CosmosDB 계정에 액세스할 수 있습니다. [자세한 정보](#)

6. [백업 정책] 탭에서 기본 설정을 유지하고 [다음]을 클릭합니다.

Azure Cosmos DB 계정 만들기 - 코어(SQL) ...

기본 사항 전역 배포 네트워크 **백업 정책** 암호화 태그 검토 + 만들기

Azure Cosmos DB는 두 가지 백업 정책을 제공합니다. 계정이 생성된 후에는 백업 정책 간에 전환할 수 없습니다. 두 백업 정책 및 가격 정보의 차이점에 대해 자세히 알아보세요. [자세한 정보](#)

백업 정책 ☒ 정기 ☐ 연속

백업 간격 분 60-1440

백업 보존 시간 8-720

보존된 데이터 복사본 2

백업 스토리지 중복 ☒ 지역 중복 백업 스토리지 ☐ 로컬 중복 백업 스토리지

7. [암호화] 탭에서 "서비스 관리형 키"를 선택하고 [검토 + 만들기]를 클릭합니다. [검토 + 만들기] 탭에서 [만들기]를 클릭합니다.

Azure Cosmos DB 계정 만들기 - 코어(SQL) ...

기본 사항 전역 배포 네트워크 백업 정책 **암호화** 태그 검토 + 만들기

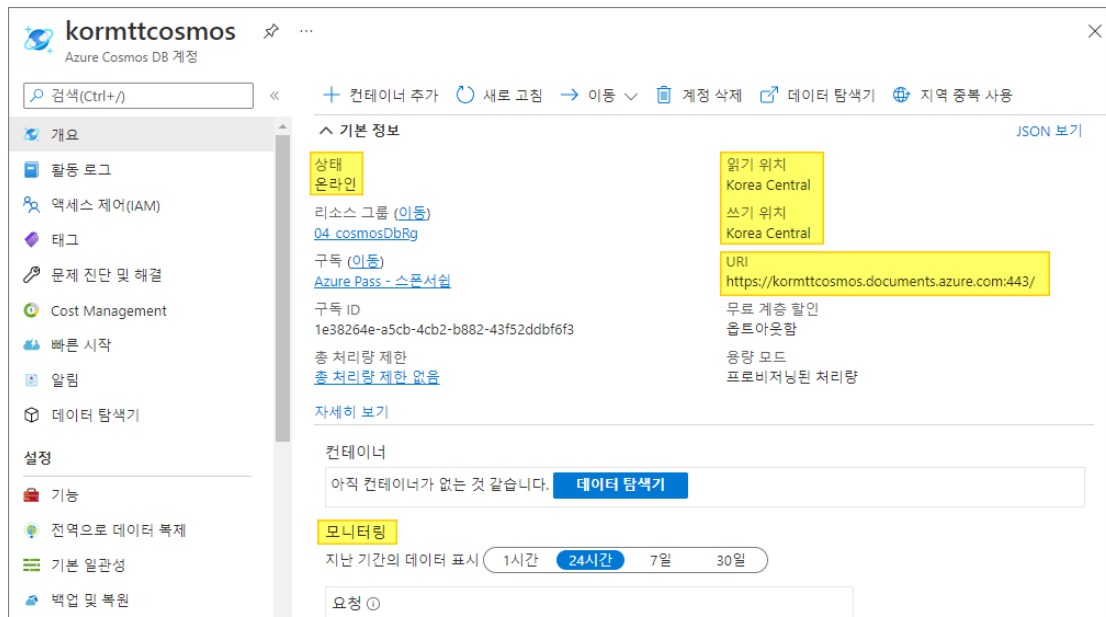
데이터 암호화

Azure Cosmos DB 암호화는 데이터 센터에서 작성되는 데이터를 원활하게 암호화하고 사용자가 데이터에 액세스할 때 자동으로 데이터를 암호 해독하여 미사용 데이터를 보호합니다.

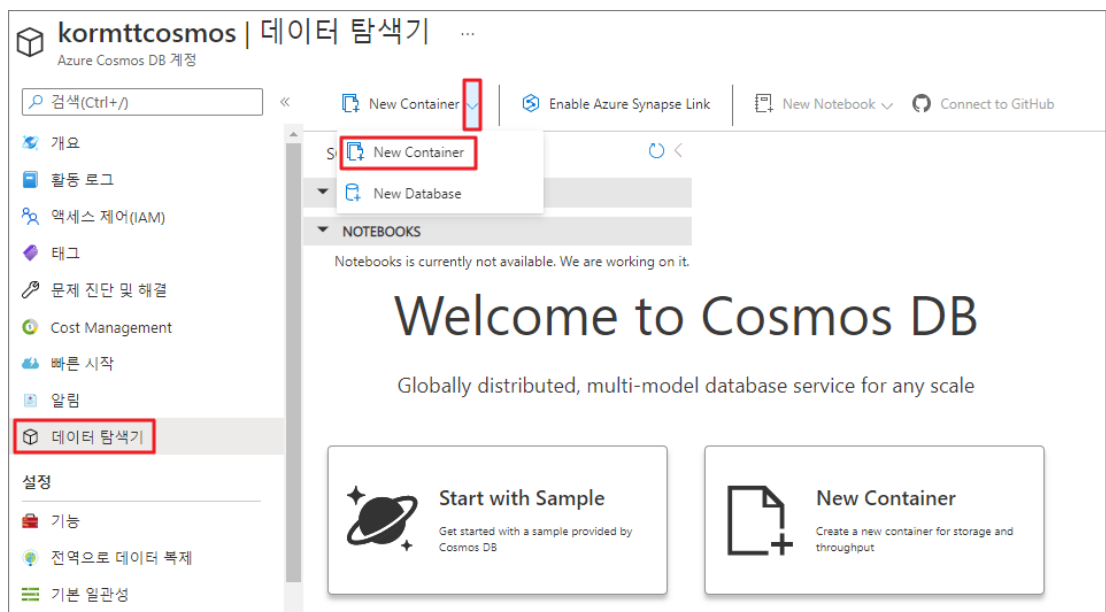
기본적으로 Azure Cosmos DB 계정은 서비스 관리형 키를 사용하여 미사용 시 암호화됩니다. 현재는 계정을 만드는 동안 고객 관리형 키를 사용하도록 선택한 후 서비스 관리형 키로 다시 전환할 수 없습니다. [자세한 정보](#)

데이터 암호화 ☒ 서비스 관리형 키 ☐ 고객 관리형 키(키 URI 입력)

8. 새로 만든 [Azure Cosmos DB 계정] 블레이드의 [개요]로 이동합니다. 다음과 같은 내용을 검토하고 [데이터 탐색기]를 클릭합니다.
- 상태: Cosmos DB의 현재 상태를 확인할 수 있습니다.
 - 읽기 위치/쓰기 위치: 전역 배포를 설정하지 않았기 때문에 Cosmos DB를 만든 위치만 표시됩니다.
 - URI: 모든 종류의 API 혹은 코드에서 Cosmos DB에 연결할 때 사용하는 URI입니다.
 - 모니터링: Cosmos DB의 시간당 사용량을 추적할 수 있기 때문에 비용을 쉽게 확인하고 월별 예상 비용을 예측할 수 있습니다.



9. 기존의 문서 탐색기, 쿼리 탐색기, 스크립트 탐색기는 이제 모두 [데이터 탐색기]를 통해 제공됩니다. [데이터 탐색기]는 데이터베이스 엔터티와 데이터베이스 스키마에 대한 작업을 제공하는 블레이드입니다. [데이터 탐색기]의 메뉴에서 [New Container - New Container]를 클릭합니다.



10. [New Container]에서 아래와 같이 구성한 후 [OK]를 클릭합니다.
- Database id: 기존 데이터베이스가 없기 때문에 "Create new"를 클릭하고 데이터베이스 이름에 "ToDoDatabase"를 입력합니다.
 - Share throughput across containers: 옵션을 체크하여 데이터베이스의 컨테이너가 처리량을 공유할 수 있도록 구성합니다.
 - Database throughput (autoscale): "Autoscale"을 선택합니다. "Manual"을 선택하여 처리량을 고정할 수도 있습니다. 처리량은 초당 RU (request unit) 메트릭으로 관리되는 성능 단위입니다.
 - Database Max RU/s: 기본 입력값을 사용합니다. 처리량 옵션을 구성하면 하단에 월별 예측 비용이

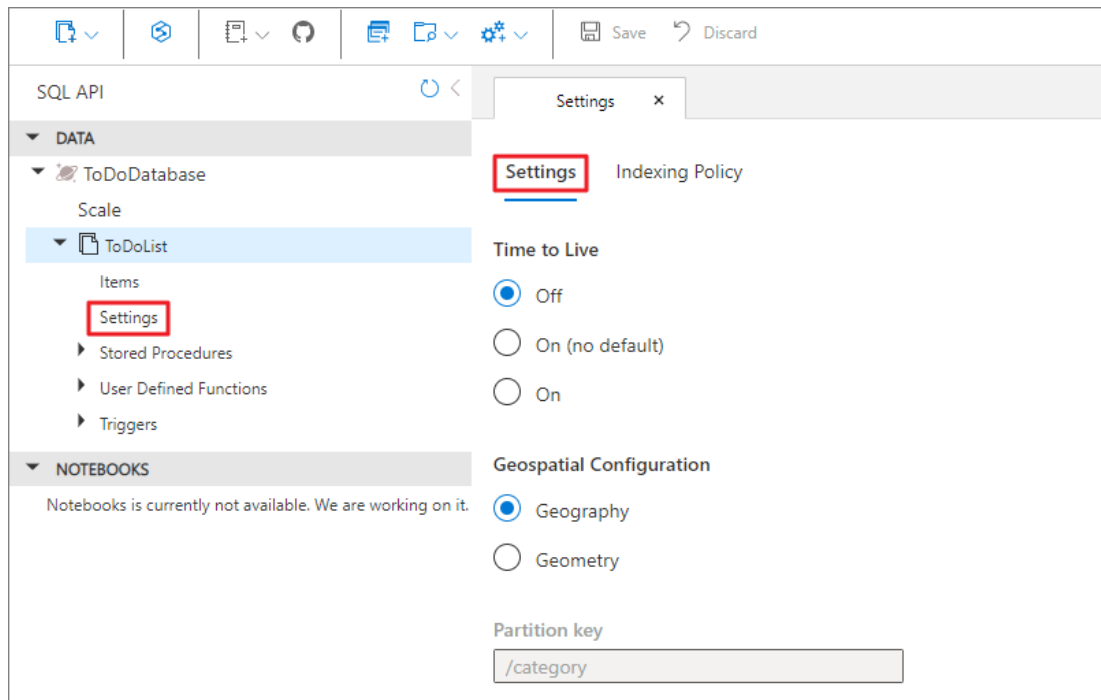
표시되는 것을 확인합니다.

- Container id: "ToDoList"를 입력합니다. 컨테이너는 데이터베이스의 테이블과 유사한 개념입니다.
- Partition key: "/category"를 입력합니다. 파티션 키는 Cosmos DB가 데이터를 논리적으로 분할하는 방법과 성능을 위해 물리적으로 분할하기 위해 사용하는 값이며 모든 엔터티에 공통적으로 포함하고 있는 값을 지정하는 것이 좋습니다.

The screenshot shows the 'New Container' configuration window. In the left pane, under 'Database id', 'Create new' is selected and 'ToDoDatabase' is entered. 'Share throughput across containers' is checked. Under 'Database throughput (autoscale)', 'Autoscale' is selected and 'Database Max RU/s' is set to 4000. The right pane shows 'Container id' as 'ToDoList' and 'Partition key' as '/category'. At the bottom right, there is an 'Enable' button for the 'Analytical store' and an 'Advanced' link.

11. [데이터 탐색기]의 네비게이션 메뉴를 검토합니다.

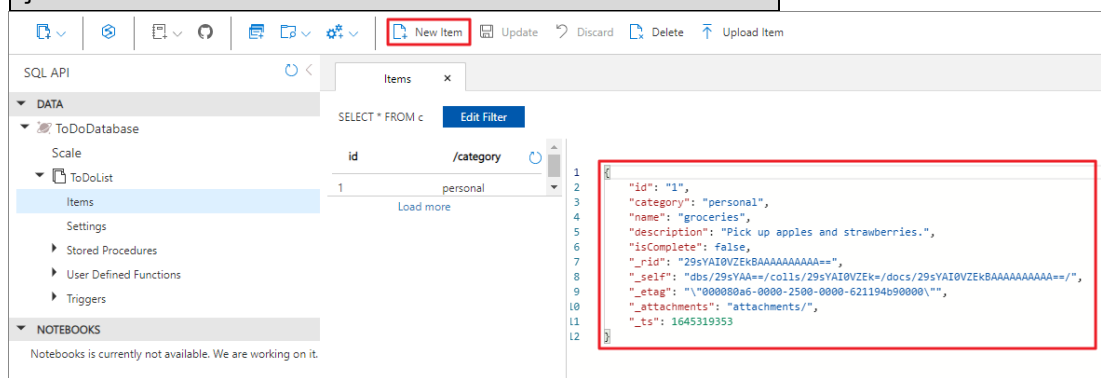
- 데이터베이스(ToDoDatabase)와 컬렉션(ToDoList)이 표시됩니다.
- Items: 컬렉션 내에 현재 있는 항목(item)의 목록이 표시됩니다.
- Settings: 데이터와 컬렉션이 유지되는 방법(TTL, Geospatial Configuration)과 인덱싱 정책을 설정할 수 있습니다.
- Stored Procedures / User Defined Functions / Triggers: Cosmos DB에서 정의할 수 있는 저장 프로시저, 사용자 정의 함수, 트리거에 대한 내용이 표시됩니다.



12. [데이터 탐색기]에서 [ToDoDatabase - ToDoList - Items]를 선택합니다. [Items] 탭에서 `SELECT * FROM c` 쿼리가 표시되며 이는 컬렉션의 모든 항목을 쿼리하겠다는 의미입니다. 메뉴에서 [New Item]을 클릭한 후 다음과 같은 JSON 데이터를 입력하고 [Save]를 클릭합니다.

- 해야 할 목록을 포함하고 있는 JSON 데이터이며 필수 요소인 `id` 값을 포함하고 있는 데이터를 입력합니다. `id`는 사용자가 설정할 수 있는 속성이며 `document`를 식별하는 고유한 이름입니다. 즉 동일한 논리 파티션 내에서 동일한 `id`를 공유하는 `document`는 없습니다.
- [Save]를 클릭하면 리소스 ID인 `_rid`, 타임스탬프를 의미하는 `_ts`, 고유한 주소를 지정할 수 있는 리소스의 URI 값인 `_self`, 항목의 변경 사항을 추적하는데 사용되는 `_etag`, 첨부 파일 리소스의 주소 지정 가능 경로인 `_attachments` 속성이 자동으로 추가되는 것을 확인할 수 있습니다.

```
{
  "id": "1",
  "category": "personal",
  "name": "groceries",
  "description": "Pick up apples and strawberries.",
  "isComplete": false
}
```



13. [데이터 탐색기]의 메뉴에서 [New Item]을 클릭한 후 다음과 같은 항목(item)을 하나 더 만들고 [Save]를 클릭합니다.

The screenshot shows the Azure Data Explorer interface. On the left, the 'DATA' pane shows the 'ToDoDatabase' and 'ToDoList' collections. The 'Items' collection is selected. In the center, a table shows two items: '1' with category 'personal' and '2' with category 'business'. On the right, the 'New Item' dialog is open, showing a JSON object for a new item:

```
{
  "id": "2",
  "category": "business",
  "name": "reminder",
  "description": "Set up meeting with Tom.",
  "isComplete": false
}
```

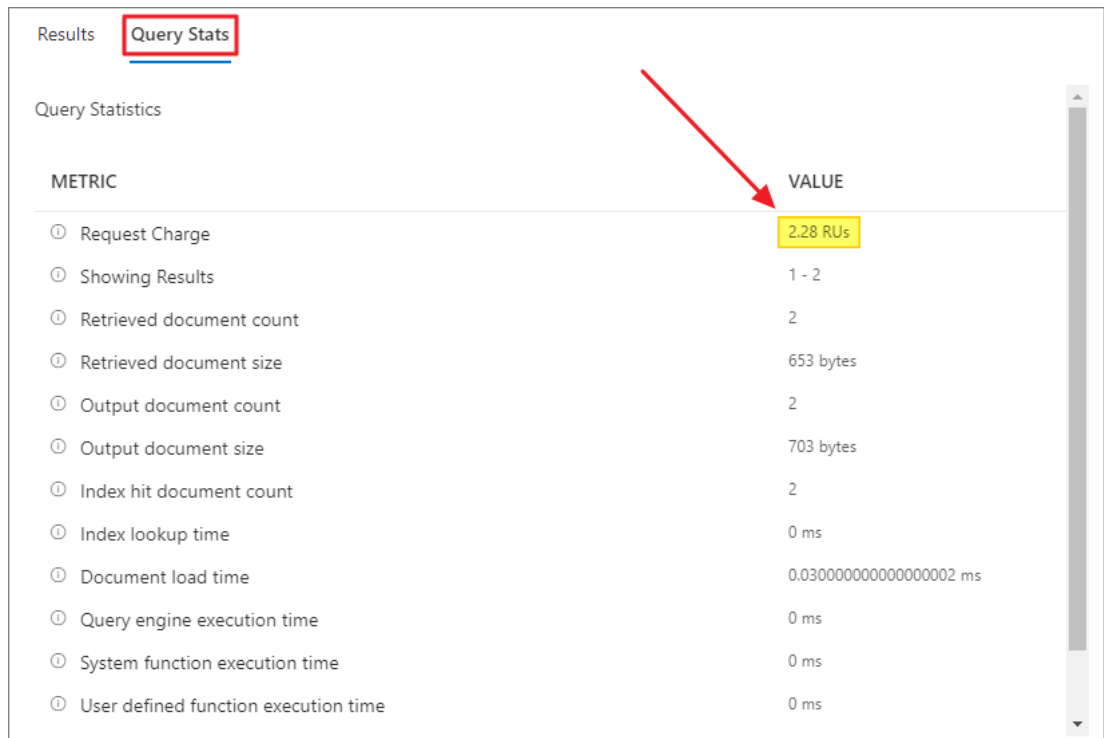
The 'New Item' button is highlighted with a red box. The JSON object is also highlighted with a red box.

14. [데이터 탐색기]의 메뉴에서 [New SQL Query]를 클릭한 후 다음과 같은 쿼리를 실행합니다.
- [Results] 탭에서 컬렉션의 모든 내용을 쿼리하여 앞서 입력했던 항목의 결과가 반환되는 것을 확인할 수 있습니다.

The screenshot shows the Azure Data Explorer interface. In the center, a new SQL query is entered: `SELECT * FROM c`. The 'Execute Query' button is highlighted with a red box. Below the query, the 'Results' tab is selected, showing the results of the query. The results are displayed as a JSON array with two items: '1' with category 'personal' and '2' with category 'business'. A red arrow points to the first item in the results.

15. 쿼리 결과의 [Query Stats] 탭으로 이동한 후 다음과 같은 내용을 검토합니다.

- Request Charge: 특정 쿼리에 얼마나 많은 RU가 사용되었는지 알려줍니다. 이 예제의 경우 2.28 RU가 사용되었습니다. 2.28 RU 중 2 RU는 2개의 항목을 읽기 위해 사용된 것이고 0.28은 데이터를 크롤링하고 이를 결합하여 반환하기 위해 사용된 것입니다.

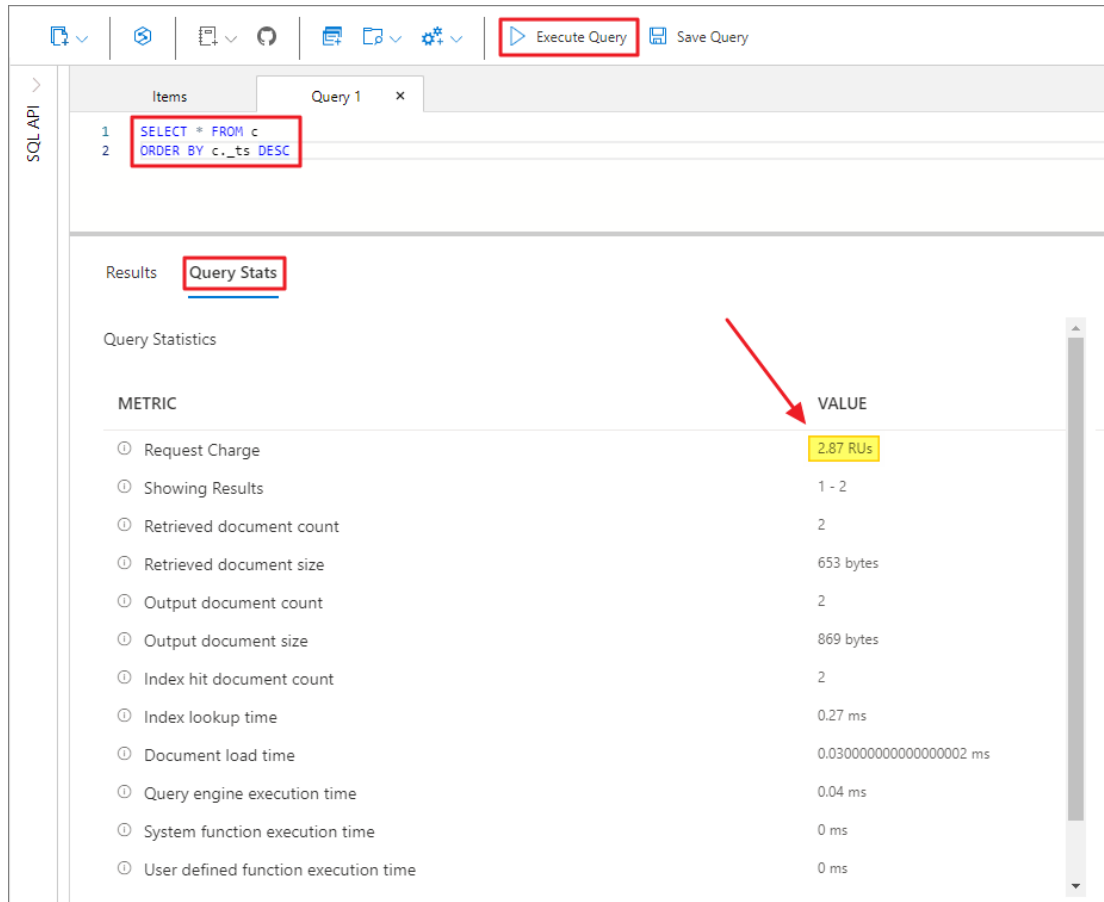


Query Statistics	
METRIC	VALUE
① Request Charge	2.28 RUs
① Showing Results	1 - 2
① Retrieved document count	2
① Retrieved document size	653 bytes
① Output document count	2
① Output document size	703 bytes
① Index hit document count	2
① Index lookup time	0 ms
① Document load time	0.030000000000000002 ms
① Query engine execution time	0 ms
① System function execution time	0 ms
① User defined function execution time	0 ms

16. 쿼리 창에서 다음 쿼리를 실행하여 항목을 타임스탬프(_ts) 기준으로 내림차순 정렬을 수행하고 [Query Stats] 탭의 내용을 검토합니다.

- 이 쿼리는 처음 실행했던 쿼리에서 타임스탬프를 기준으로 내림차순 정렬을 수행하기 때문에 더 많은 RU가 사용되었음을 확인할 수 있습니다.

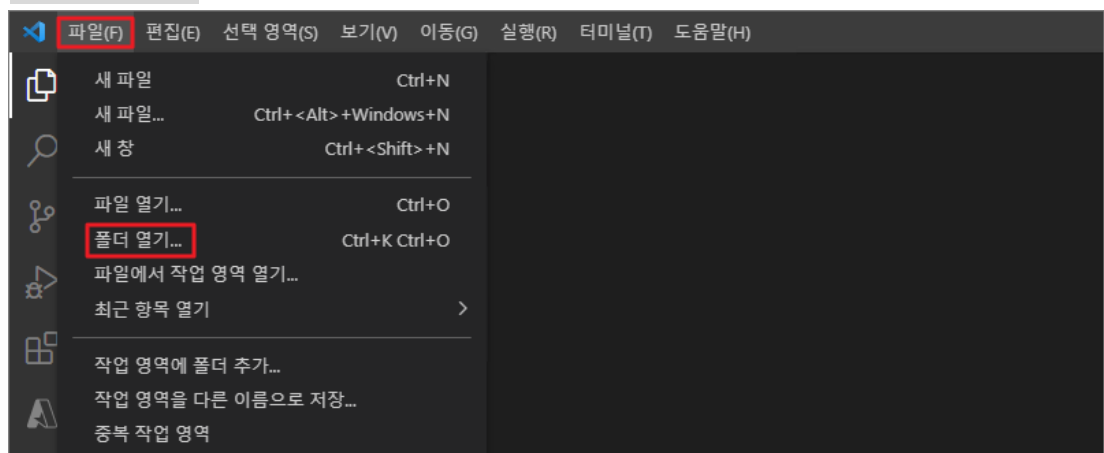
```
SELECT * FROM c
ORDER BY c._ts DESC
```



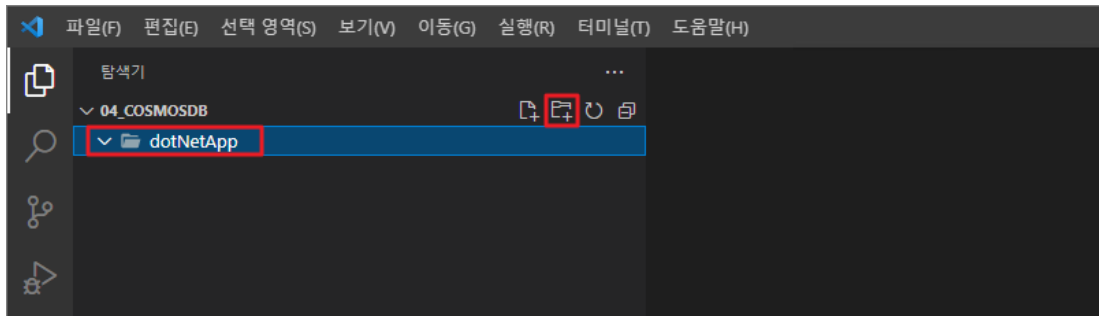
TASK 02. Visual Studio Code에서 Cosmos DB 개발

이 작업에서는 .NET에서 Azure Cosmos DB에 연결하는 간단한 개발 환경을 실습합니다.

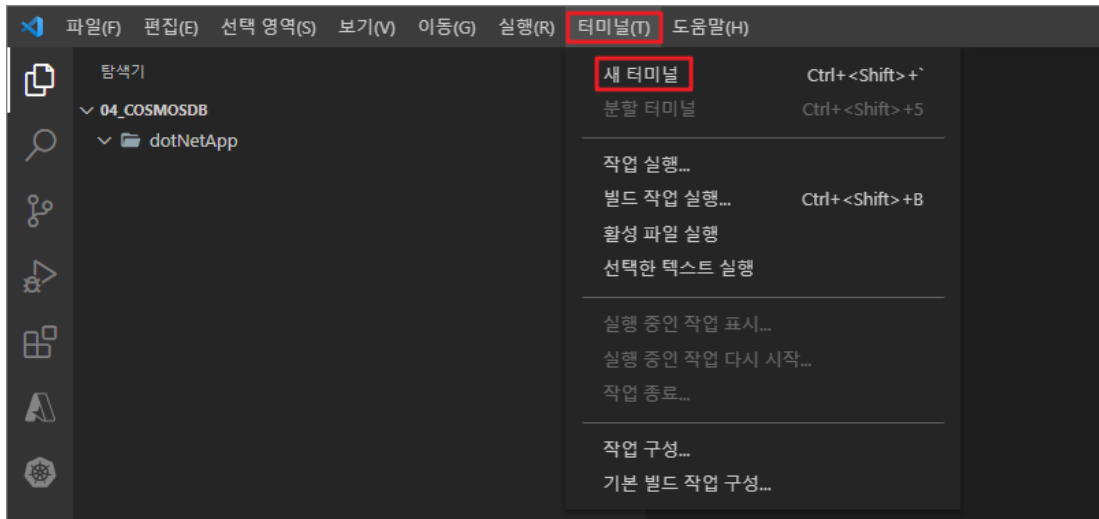
1. labVM 가상 머신에 로그인 합니다. Visual Studio Code를 열고 메뉴에서 [파일 - 폴더 열기]를 클릭합니다. C:\04_cosmosDb 폴더를 만들고 이 폴더를 선택합니다.



2. Visual Studio Code의 [탐색기]에서 [새 폴더] 아이콘을 클릭하고 "dotNetApp" 이름의 폴더를 만듭니다.



3. Visual Studio Code의 메뉴에서 [터미널 - 새 터미널]을 클릭합니다.



4. [터미널] 창에서 다음 명령을 실행하여 새 콘솔 애플리케이션을 빌드합니다.

```
# 새 콘솔 애플리케이션 만들기
cd dotNetApp
dotnet new console
```

```
문제 출력 디버그 콘솔 터미널

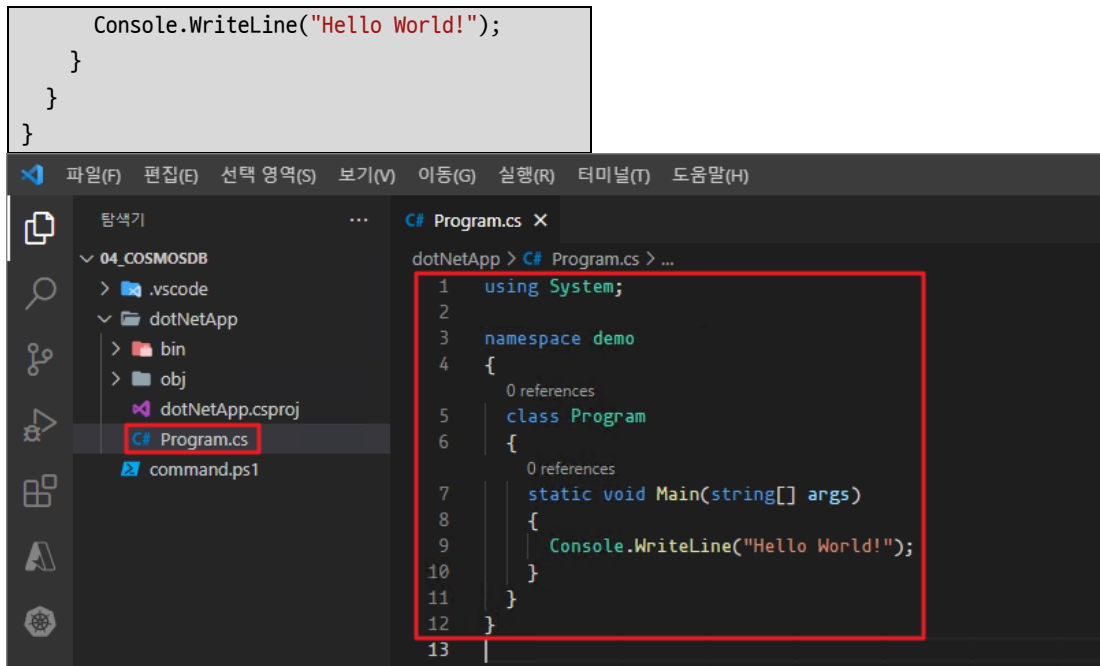
PS C:\04_cosmosDb> # 새 콘솔 애플리케이션 만들기
PS C:\04_cosmosDb> cd dotNetApp
PS C:\04_cosmosDb\dotNetApp> dotnet new console
"콘솔 앱" 템플릿이 성공적으로 생성되었습니다.

생성 후 작업 처리 중...
C:\04_cosmosDb\dotNetApp\dotNetApp.csproj에서 'dotnet restore' 실행 중 ...
복원할 프로젝트를 확인하는 중...
C:\04_cosmosDb\dotNetApp\dotNetApp.csproj을(를) 142 ms 동안 복원했습니다.
복원에 성공했습니다.

PS C:\04_cosmosDb\dotNetApp> |
```

5. Visual Studio Code의 [탐색기]에서 `C:\04_cosmosDb\dotNetApp\Program.cs` 파일을 열고 다음과 같은 기본 콘솔 애플리케이션 코드를 입력합니다.

```
using System;
namespace demo
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```



6. Visual Studio Code의 [터미널] 창에서 다음 명령을 실행하여 Azure Cosmos DB 패키지를 추가합니다.

Cosmos DB 패키지 추가

dotnet add package Microsoft.Azure.Cosmos

```

문제 출력 디버그 콘솔 터미널

PS C:\04_cosmosDb\dotNetApp> # Cosmos DB 패키지 추가
PS C:\04_cosmosDb\dotNetApp> dotnet add package Microsoft.Azure.Cosmos
복원할 프로젝트를 확인하는 중...
Writing C:\Users\labAdmin\AppData\Local\Temp\tmp87EF.tmp
info : 'C:\04_cosmosDb\dotNetApp\dotNetApp.csproj' 프로젝트에 'Microsoft.Azure.Cosmos' 패키지에 대한 PackageReference를 추가하...
info : GET https://api.nuget.org/v3/registration5-gz-semver2/microsoft.azure.cosmos/index.json 822밀리초
info : OK https://api.nuget.org/v3/registration5-gz-semver2/microsoft.azure.cosmos/index.json 822밀리초
info : C:\04_cosmosDb\dotNetApp\dotNetApp.csproj의 패키지를 복원하는 중...
info : GET https://api.nuget.org/v3-flatcontainer/microsoft.azure.cosmos/index.json 812밀리초
info : OK https://api.nuget.org/v3-flatcontainer/microsoft.azure.cosmos/index.json 812밀리초
info : GET https://api.nuget.org/v3-flatcontainer/microsoft.azure.cosmos/3.26.1/microsoft.azure.cosmos.3.26.1.nupkg 48밀리초
info : OK https://api.nuget.org/v3-flatcontainer/microsoft.azure.cosmos/3.26.1/microsoft.azure.cosmos.3.26.1.nupkg 48밀리초
info : GET https://api.nuget.org/v3-flatcontainer/azure.core/index.json
info : GET https://api.nuget.org/v3-flatcontainer/microsoft.bcl.asyncinterfaces/index.json
info : GET https://api.nuget.org/v3-flatcontainer/microsoft.bcl.hashcode/index.json
info : GET https://api.nuget.org/v3-flatcontainer/system.buffers/index.json

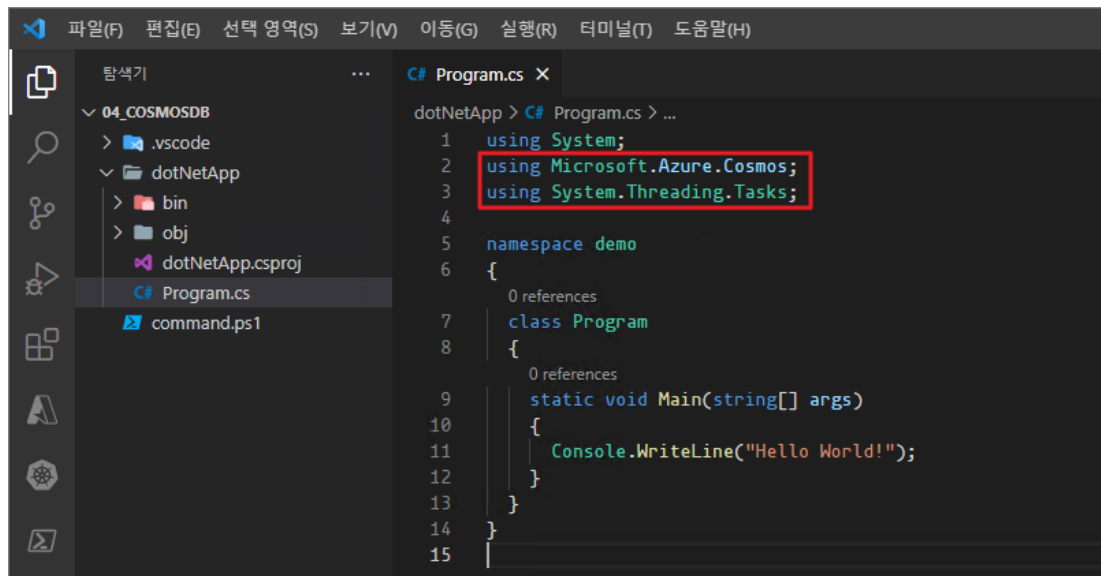
```

7. Visual Studio Code의 [탐색기]에서 C:\04_cosmosDb\dotNetApp\Program.cs 파일을 열고 다음과 같은 using 문을 추가합니다.

```

using Microsoft.Azure.Cosmos;
using System.Threading.Tasks;

```

8. `static void Main` 아래에 다음과 같은 코드를 추가합니다.

- ① `CosmosClient client`: URL과 Cosmos DB 키를 사용하여 Cosmos DB 클라이언트를 초기화합니다.
- ② `Database database`: 데이터베이스가 없는 경우 `var databaseName`에서 정의한 이름의 데이터베이스를 생성합니다.
- ③ `Container container`: 없는 경우 `MyContainerName` 이름의 컨테이너를 만들고 `partitionKeyPath` 이름의 파티션 키를 만들고 400 RU를 할당합니다.
- ④ `dynamic testItem`: ID, `partitionKeyPath`, `details` 정보가 있는 동적 개체를 생성합니다.
- ⑤ `var response`: 컨테이너를 호출하고 해당 항목(item)을 생성합니다.

```
private static async Task CreateItem()
{
    var cosmosUrl = "";
    var cosmoskey = "";
    var databaseName = "";
    CosmosClient client = new CosmosClient(cosmosUrl, cosmoskey);
    Database database = await client.CreateDatabaseIfNotExistsAsync(databaseName);
    Container container = await database.CreateContainerIfNotExistsAsync(
        "MyContainerName", "/partitionKeyPath", 400);

    dynamic testItem = new { id = Guid.NewGuid().ToString(), partitionKeyPath =
        "MyTestPkValue", details = "it's working" };
    var response = await container.CreateItemAsync(testItem);
}
```

```

C# Program.cs X
dotNetApp > C# Program.cs > ...

0 references
9 static void Main(string[] args)
10 {
11     CreateItem().Wait();
12 }
13

1 reference
14 private static async Task CreateItem()
15 {
16     var cosmosUrl = "";
17     var cosmoskey = "";
18     var dbName = "";
19
20     CosmosClient client = new CosmosClient(cosmosUrl, cosmoskey);
21     Database database = await client.CreateDatabaseIfNotExistsAsync(dbName);
22     Container container = await database.CreateContainerIfNotExistsAsync(
23         "MyContainerName", "/partitionKeyPath", 400);
24
25     dynamic testItem = new { id = Guid.NewGuid().ToString(), partitionKeyPath = "MyTestPkValue", details = "it's working" };
26     var response = await container.CreateItemAsync(testItem);
27 }
28 }
29
30

```

9. Console.WriteLine 부분을 다음과 같은 코드로 변경합니다.

```

static void Main(string[] args)
{
    CreateItem().Wait();
}

```

```

C# Program.cs X
dotNetApp > C# Program.cs > ...

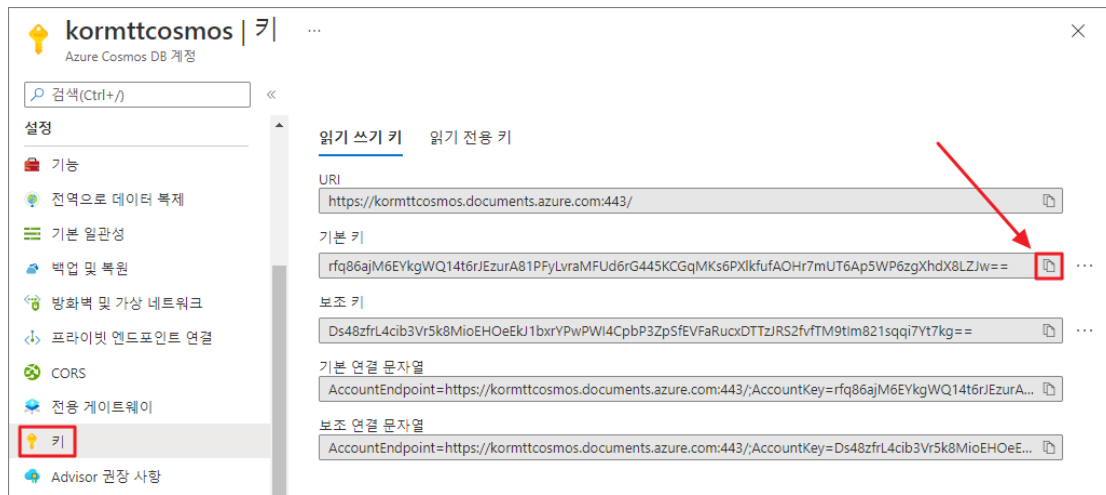
1 using System;
2 using Microsoft.Azure.Cosmos;
3 using System.Threading.Tasks;
4
5 namespace demo
6 {
7     0 references
8     class Program
9     {
10         0 references
11         static void Main(string[] args)
12         {
13             CreateItem().Wait();
14         }
15     }
16
17     1 reference
18     private static async Task CreateItem()

```

10. 이제 애플리케이션에서 사용할 Azure Cosmos DB의 URL과 키를 확인해야 합니다. [Azure Cosmos DB 계정] 블레이드의 [개요]로 이동한 후 URI 값을 메모장에 복사합니다.

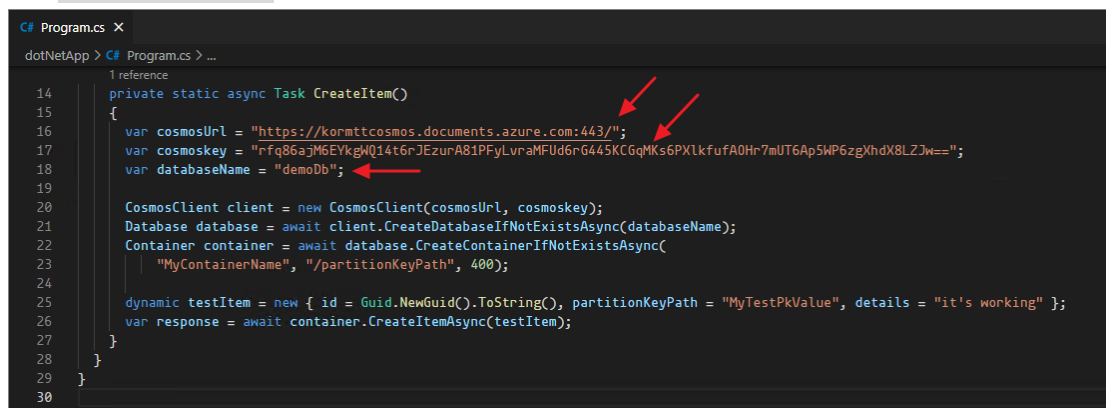


11. [Azure Cosmos DB 계정] 블레이드의 [설정 - 키]로 이동한 후 "기본 키" 값을 메모장에 복사합니다.

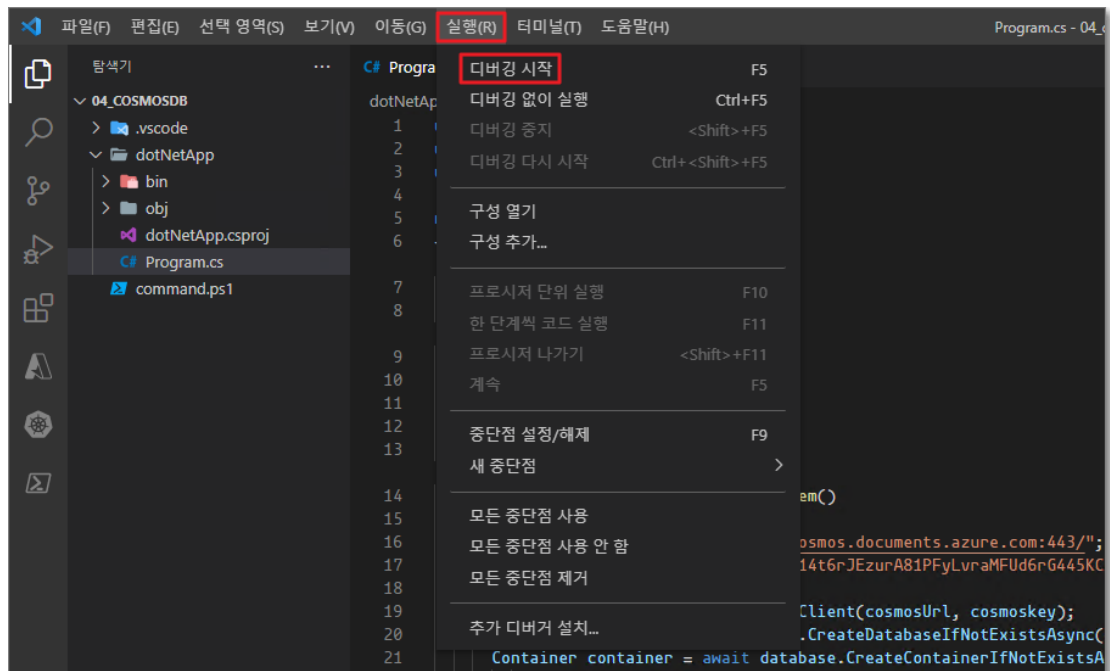


12. Visual Studio Code로 전환한 후 변수 선언 부분을 다음과 같이 수정하고 변경 내용을 저장합니다.

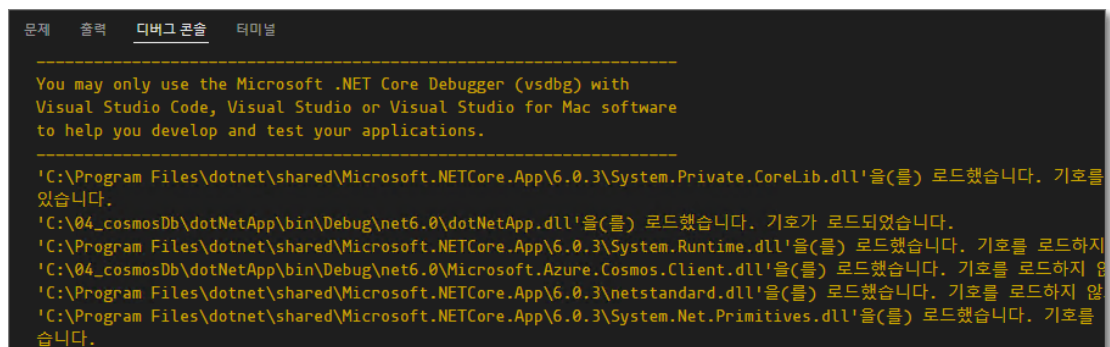
- `var cosmosUrl`: 메모장에 복사했던 Cosmos DB의 URI 값을 붙여 넣습니다.
- `var cosmosKey`: 메모장에 복사했던 Cosmos DB의 기본 키 값을 붙여 넣습니다.
- `var databaseName`: 새로 만들 데이터베이스 이름을 "demoDb"로 지정합니다.



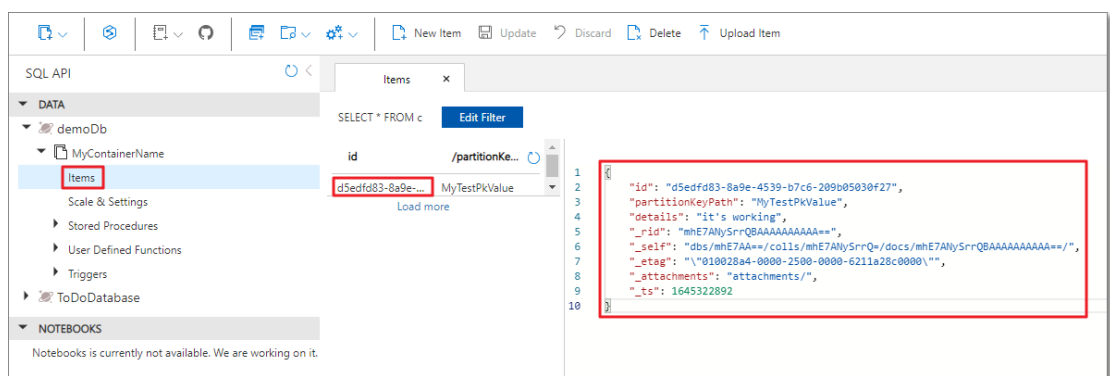
13. Visual Studio Code의 메뉴에서 [실행 - 디버깅 시작]을 클릭하거나 F5 키를 누릅니다.



14. [디버그 콘솔] 탭에서 오류 없이 디버깅이 완료된 것을 확인합니다.



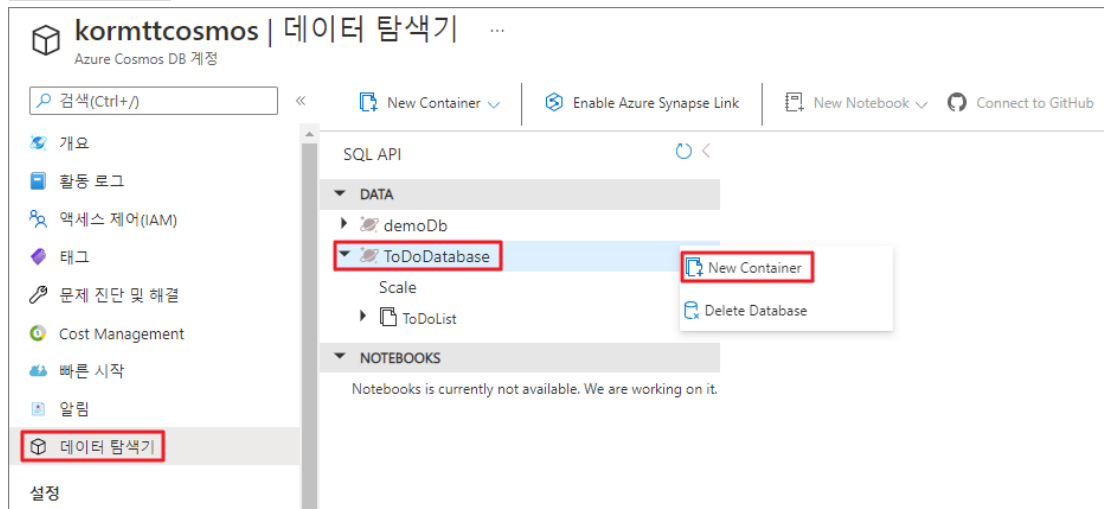
15. [Azure Cosmos DB 계정] 블레이드로 전환한 후 [데이터 탐색기]를 클릭합니다. [데이터 탐색기]의 네비게이션 메뉴에서 [demoDb - MyContainerName - Items]로 이동한 후 생성된 항목을 클릭하고 Visual Studio Code에서 입력했던 내용이 표시되는 것을 확인합니다. 즉 애플리케이션을 통해 Cosmos DB를 만들고 RU를 설정하고 컨테이너를 만들고 항목을 만드는 작업을 쉽게 진행할 수 있습니다.



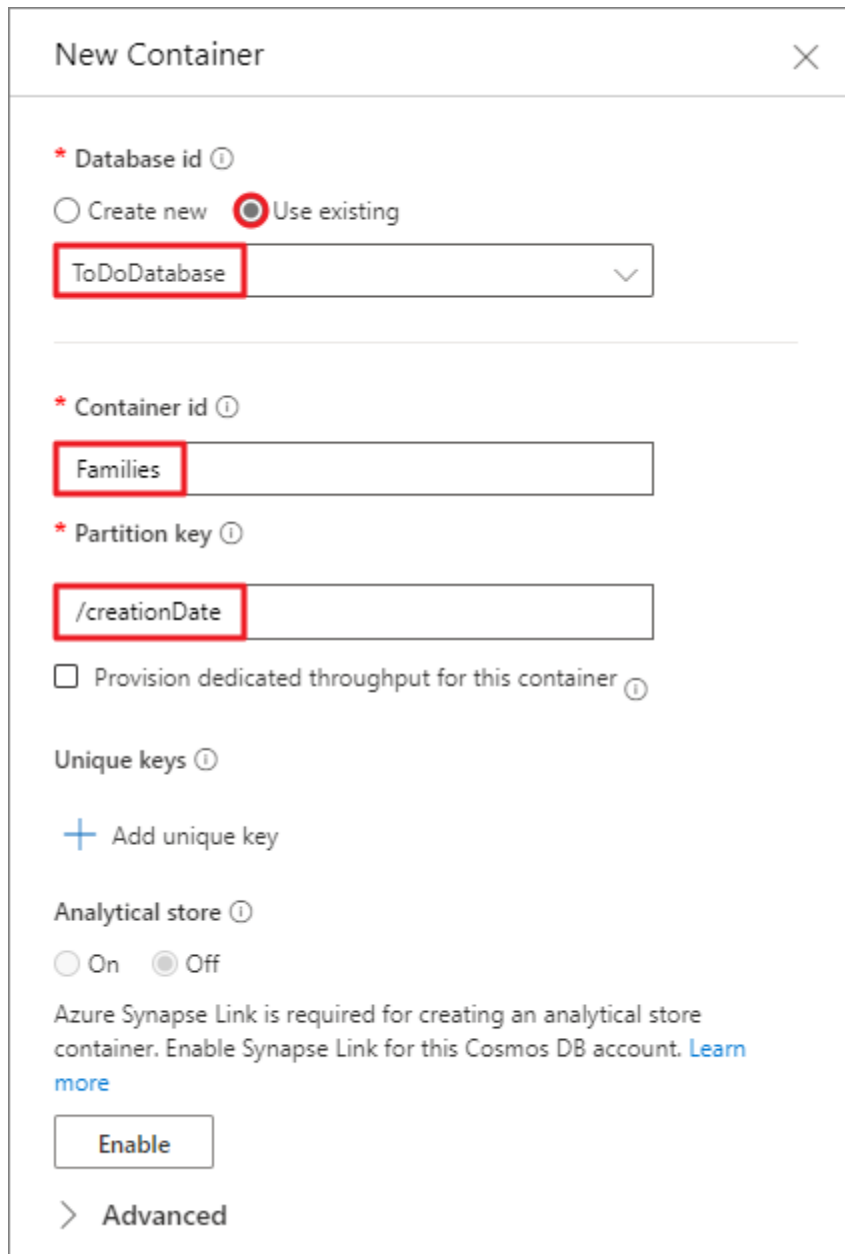
TASK 03. Cosmos DB 쿼리

이 작업에서는 Cosmos DB에 새 데이터베이스를 만들고 이에 대한 쿼리를 실행합니다.

1. [Azure Cosmos DB 계정] 블레이드의 [데이터 탐색기]로 이동합니다. 네비게이션에서 [DATA - ToDoDatabase]를 선택하고 [... - New Container]를 클릭합니다.



2. [New Container]에서 아래와 같이 구성하고 [OK]를 클릭합니다.
 - Database id: "Use existing"을 선택하고 **ToDoDatabase**를 선택합니다.
 - Container id: Families
 - Partition key: /creationDate



New Container

* Database id ⓘ

☐ Create new ☒ Use existing

ToDoDatabase

* Container id ⓘ

Families

* Partition key ⓘ

/creationDate

☐ Provision dedicated throughput for this container ⓘ

Unique keys ⓘ

+ Add unique key

Analytical store ⓘ

☐ On ☒ Off

Azure Synapse Link is required for creating an analytical store container. Enable Synapse Link for this Cosmos DB account. [Learn more](#)

Enable

> Advanced

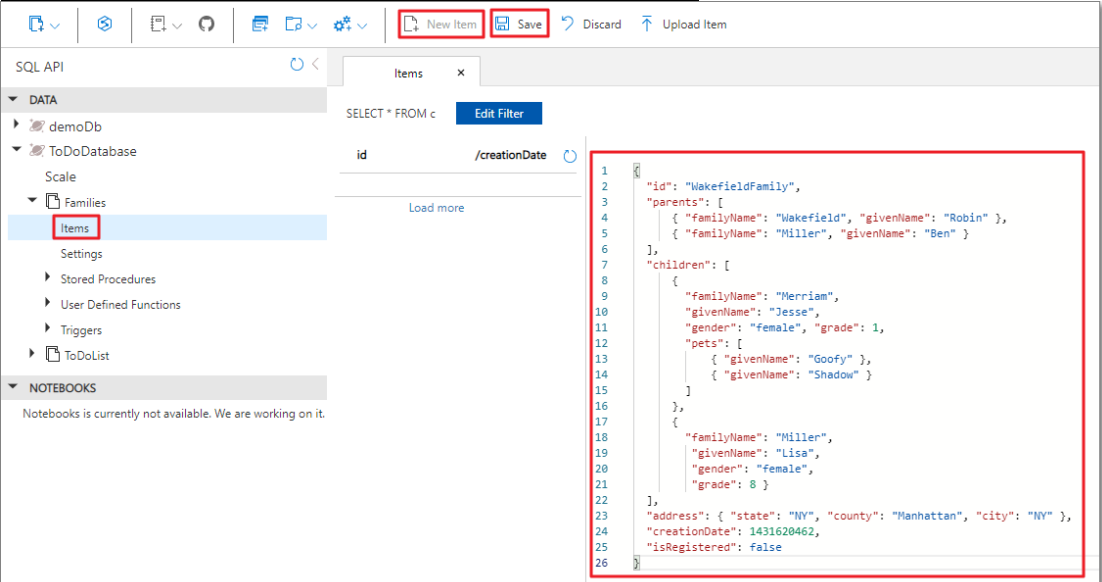
3. [데이터 탐색기]의 [DATA - ToDoDatabase - Families - Items]를 선택한 후 메뉴에서 [New Item]을 클릭합니다. 아래와 같은 JSON 내용을 입력하고 [Save]를 클릭합니다.

```
{
  "id": "WakefieldFamily",
  "parents": [
    { "familyName": "Wakefield", "givenName": "Robin" },
    { "familyName": "Miller", "givenName": "Ben" }
  ],
  "children": [
    {
      "familyName": "Merriam",
      "givenName": "Jesse",
    }
  ]
}
```

```

    "gender": "female", "grade": 1,
    "pets": [
      { "givenName": "Goofy" },
      { "givenName": "Shadow" }
    ]
  },
  {
    "familyName": "Miller",
    "givenName": "Lisa",
    "gender": "female",
    "grade": 8 }
],
"address": { "state": "NY", "county": "Manhattan",
"city": "NY" },
"creationDate": 1431620462,
"isRegistered": false
}

```



```

1  {
2    "id": "WakefieldFamily",
3    "parents": [
4      { "familyName": "Wakefield", "givenName": "Robin" },
5      { "familyName": "Miller", "givenName": "Ben" }
6    ],
7    "children": [
8      {
9        "familyName": "Merriam",
10       "givenName": "Jesse",
11       "gender": "female", "grade": 1,
12       "pets": [
13         { "givenName": "Goofy" },
14         { "givenName": "Shadow" }
15       ]
16     },
17     {
18       "familyName": "Miller",
19       "givenName": "Lisa",
20       "gender": "female",
21       "grade": 8 }
22   ],
23   "address": { "state": "NY", "county": "Manhattan", "city": "NY" },
24   "creationDate": 1431620462,
25   "isRegistered": false
26 }

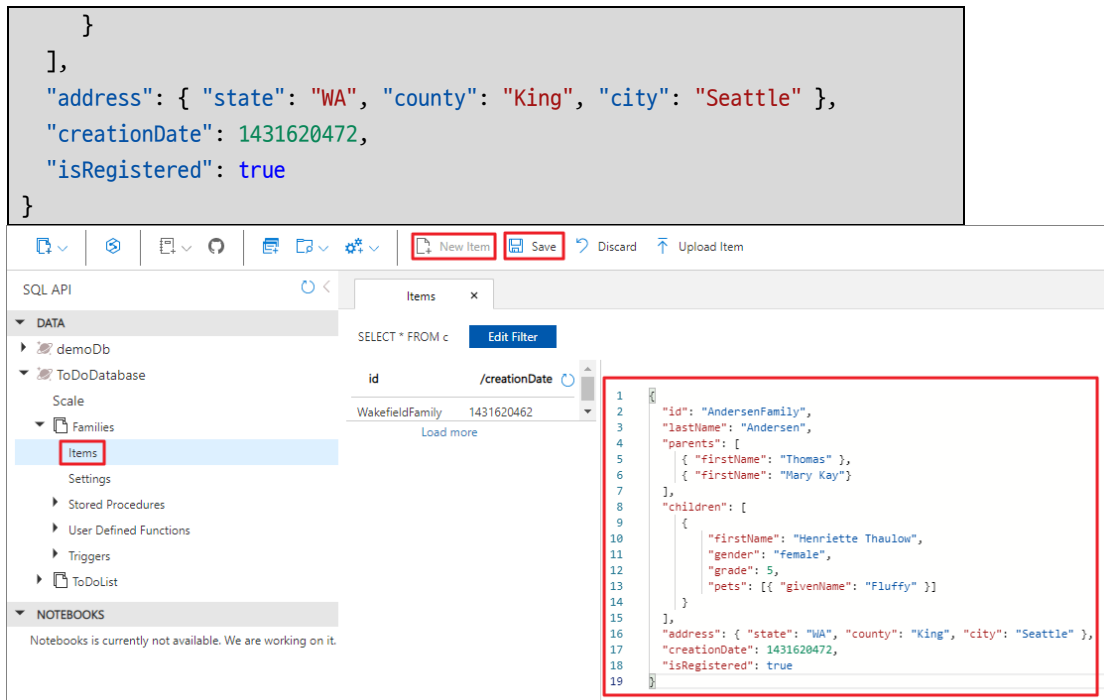
```

4. [데이터 탐색기]에서 다시 [New Item]을 클릭한 후 아래와 같은 JSON 내용을 입력하고 [Save]를 클릭합니다.

```

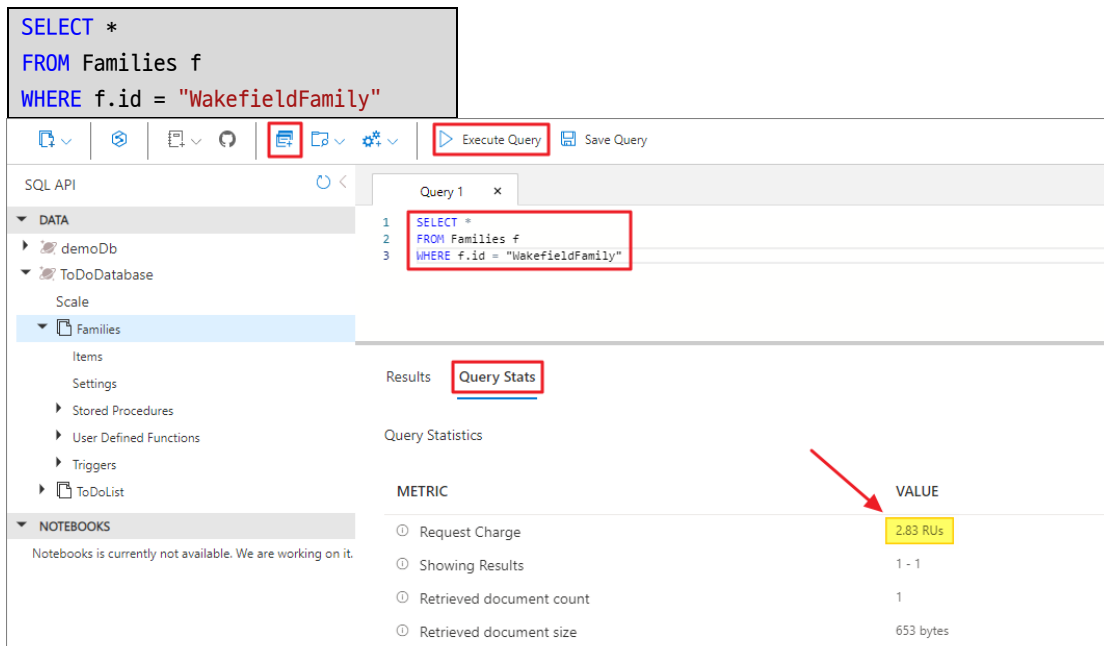
{
  "id": "AndersenFamily",
  "lastName": "Andersen",
  "parents": [
    { "firstName": "Thomas" },
    { "firstName": "Mary Kay" }
  ],
  "children": [
    {
      "firstName": "Henriette Thaulow",
      "gender": "female",
      "grade": 5,
      "pets": [{ "givenName": "Fluffy" }]
    }
  ]
}

```



5. 이제 입력한 항목에 대한 쿼리를 실행합니다. [데이터 탐색기]의 메뉴에서 [New SQL Query]를 클릭합니다. 쿼리창에 다음과 같은 쿼리를 입력한 후 실행합니다.

- WHERE 절에서 Families 테이블의 ID를 기준으로 조건을 정의하여 쿼리를 실행합니다.
- [Query Stats] 탭에서 요청 비용(Request Charge)을 확인하면 생각보다 많은 RU가 사용되었음을 확인할 수 있습니다.
- 선택한 ID가 파티션에 없기 때문에 쿼리 비용이 효율적이지 않은 것을 확인할 수 있습니다.



6. [데이터 탐색기]에서 [New SQL Query]를 클릭하고 쿼리창에서 다음 쿼리를 실행합니다.
- Families 컬렉션의 children에 대한 givenName 값만 출력되는 것을 확인할 수 있습니다.

```
SELECT c.givenName
```



```

FROM Families f
JOIN c IN f.children
WHERE f.id = 'WakefieldFamily'

```

SQL API

DATA

- demoDb
- ToDoDatabase
 - Scale
 - Families
 - Items
 - Settings
 - Stored Procedures
 - User Defined Functions
 - Triggers
 - ToDoList

NOTEBOOKS

Notebooks is currently not available. We are working on it.

Query 1 x

```

1 SELECT c.givenName
2 FROM Families f
3 JOIN c IN f.children
4 WHERE f.id = 'WakefieldFamily'

```

Execute Query Save Query

Results Query Stats

1 - 2

```

{
  "givenName": "Jesse"
},
{
  "givenName": "Lisa"
}

```

7. 쿼리창에서 다음 쿼리를 실행합니다. Cosmos DB에서는 이와 같은 쿼리를 통해 개체(object)를 생성할 수 있기 때문에 일반적인 쿼리보다 훨씬 강력합니다.

```

SELECT {'Name':f.id, 'City':f.address.city} AS Family
FROM Families f
WHERE f.address.city = f.address.state

```

SQL API

DATA

- demoDb
- ToDoDatabase
 - Scale
 - Families
 - Items
 - Settings
 - Stored Procedures
 - User Defined Functions
 - Triggers
 - ToDoList

NOTEBOOKS

Notebooks is currently not available. We are working on it.

Query 1 x

```

1 SELECT {'Name': f.id, 'City': f.address.city} AS Family
2 FROM Families f
3 WHERE f.address.city = f.address.state

```

Execute Query Save Query

Results Query Stats

1 - 1

```

{
  "Family": {
    "Name": "WakefieldFamily",
    "City": "NY"
  }
}

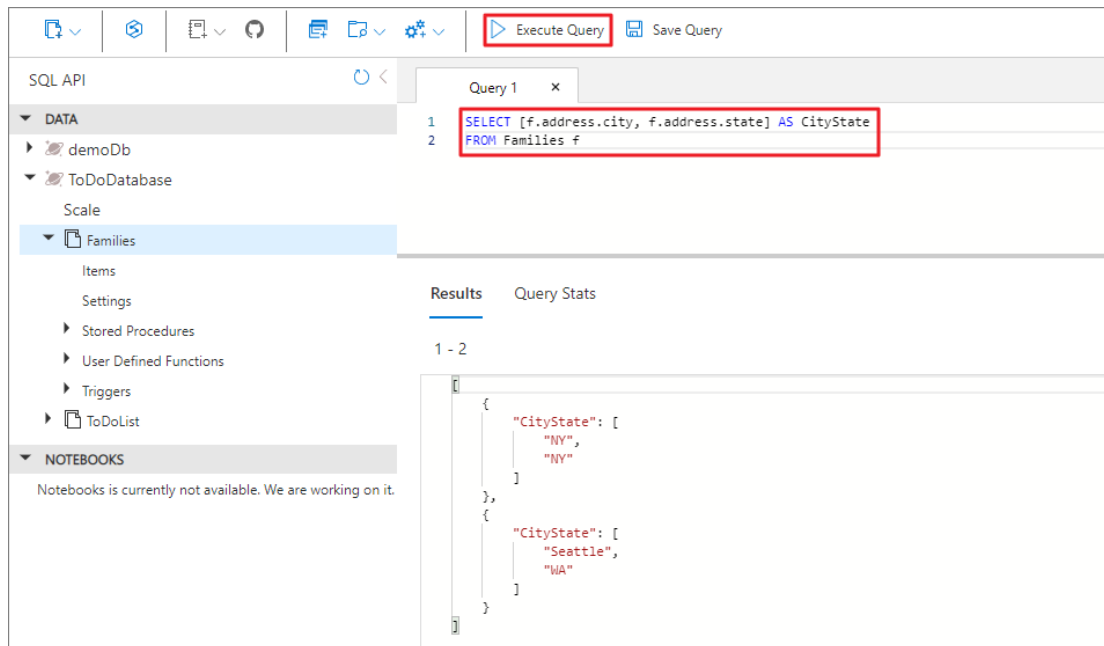
```

8. 쿼리 창에서 다음 쿼리를 실행합니다. Cosmos DB는 개체(object) 외에도 배열(array)을 수행하여 정보를 나열할 수 있습니다. 다음 쿼리를 실행하면 모든 항목의 city와 state를 표시해줍니다.

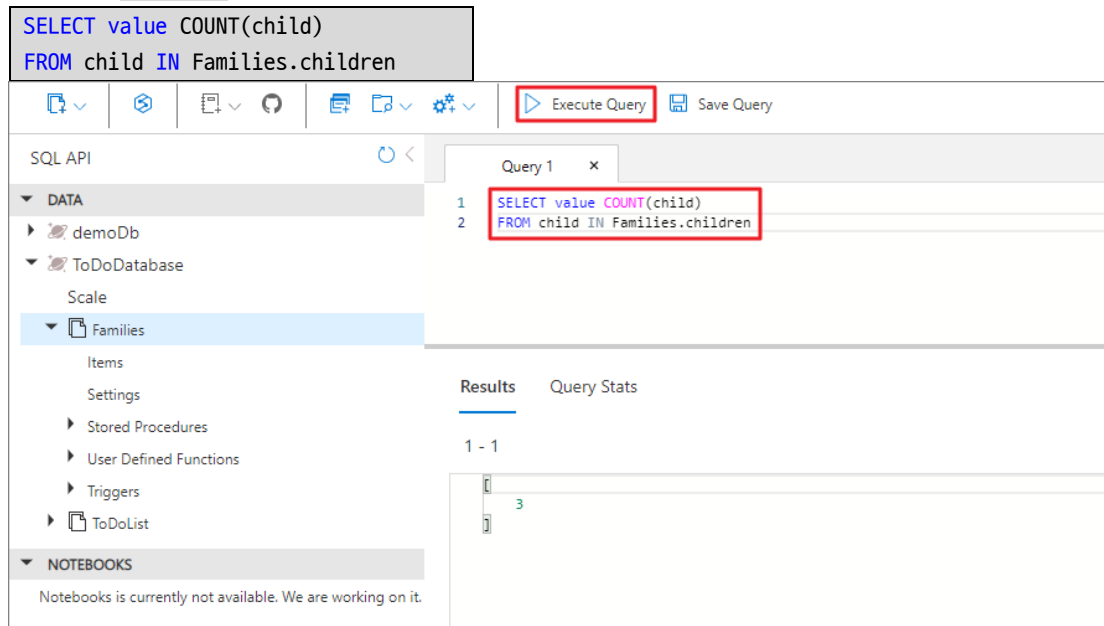
```

SELECT [f.address.city, f.address.state] AS CityState
FROM Families f

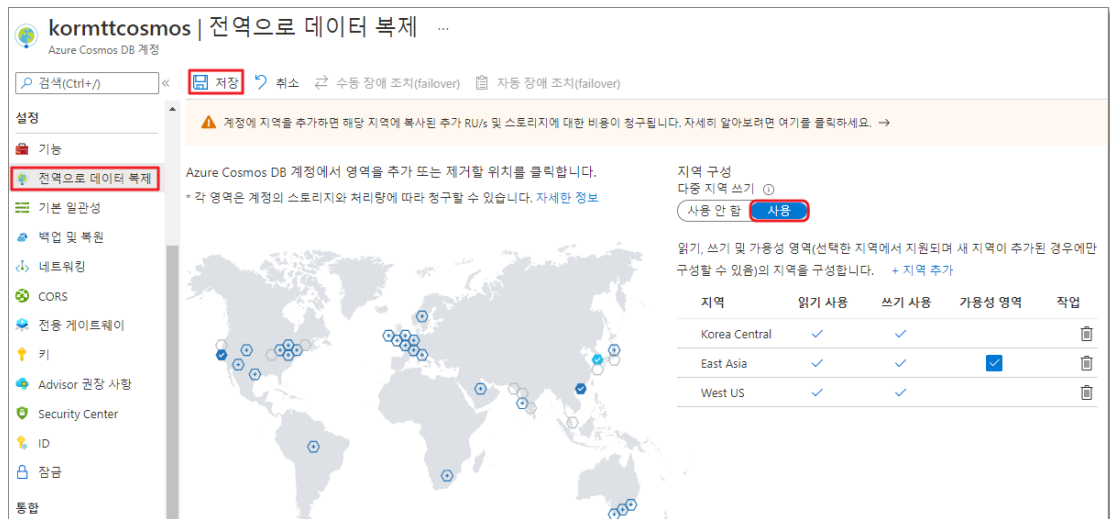
```



9. 쿼리창에서 다음과 같은 쿼리를 실행합니다. Cosmos DB에서는 집계를 수행할 수 있으며 이 쿼리의 경우 각 항목의 `children` 수를 확인할 수 있습니다.

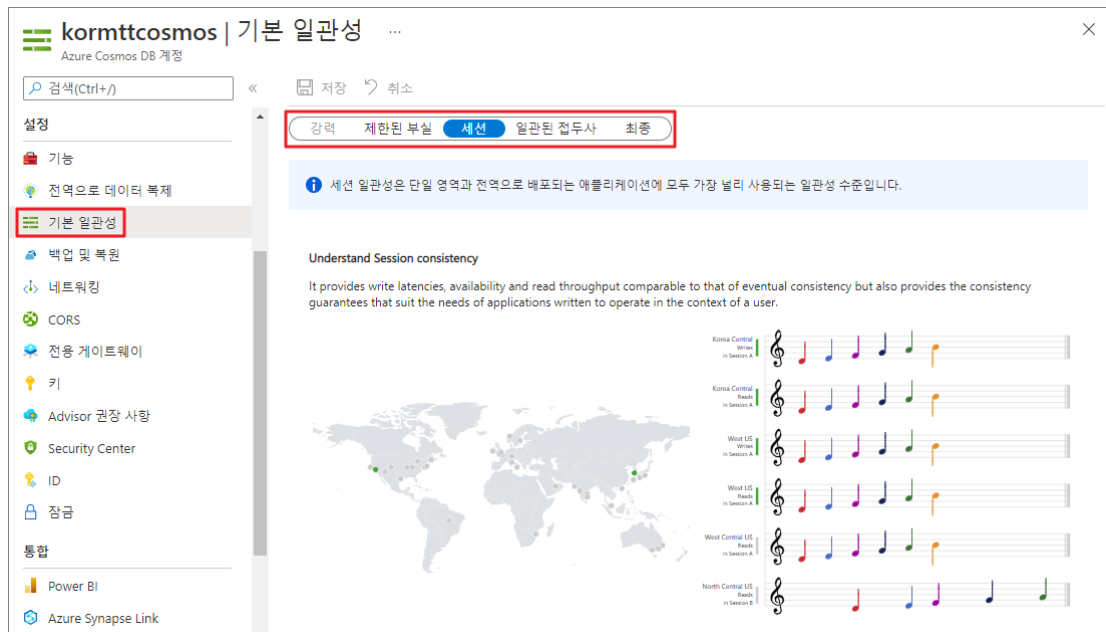


10. [Azure Cosmos DB 계정] 블레이드의 [설정 - 전역으로 데이터 복제]로 이동합니다. 이 설정에서 데이터를 복제할 지역을 선택하고 다중 지역 쓰기(multi-region write) 옵션을 활성화할 수 있습니다. 원하는 지역을 선택하고 다중 지역 쓰기 옵션을 [사용]으로 설정한 후 [저장]을 클릭합니다.



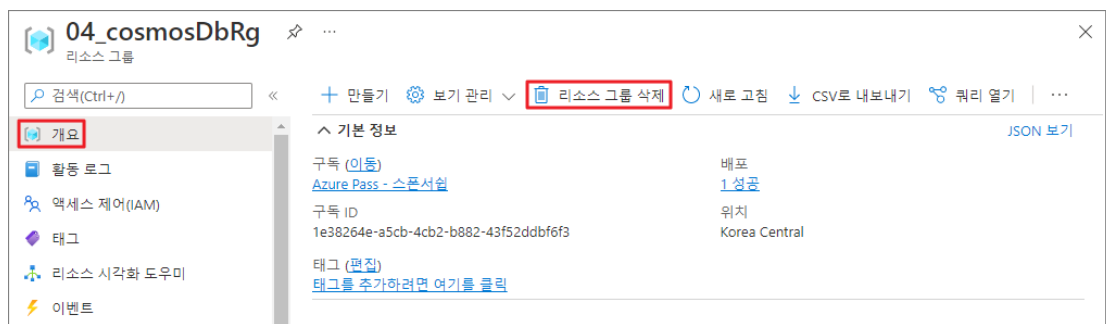
11. [Azure Cosmos DB 계정] 블레이드의 [설정 - 기본 일관성]으로 이동합니다. 다음과 같은 일관성 모델을 확인할 수 있습니다.

- **강력(strong)**: 선형화 가능성(linearizability)을 보장합니다. 선형화 가능성은 요청을 동시에 처리하는 것을 의미합니다. 읽기는 항목(item)의 가장 최근에 커밋된 버전을 반환하도록 보장됩니다. 클라이언트는 커밋되지 않은 쓰거나 부준적인 쓰기를 절대 볼 수 없기 때문에 사용자는 항상 최신에 커밋된 쓰기를 읽을 수 있습니다.
- **제한된 부실(bounded staleness)**: 읽기는 일관된 접두사(consistent-prefix) 보장을 준수하도록 보장되지만 항목의 "K" 버전(업데이트) 혹은 "T" 시간 간격 중 먼저 도달하는 시간만큼 쓰기보다 지연될 수 있습니다. 즉 항목의 버전 수(K), 읽기가 쓰기보다 뒤쳐질 수 있는 시간 간격(T)이라는 두 가지 방식으로 설정할 수 있습니다. **bounded stateless**는 낮은 대기 시간을 유지하면서 전체 전역 순서를 보장해야 하는 글로벌하게 배포되는 애플리케이션에서 자주 선택됩니다.
- **세션 일관성(session consistency)**: 단일 클라이언트 세션 내의 읽기에서 일관적인 접두사(consistent-prefix), 단조 읽기(monotonic read), 단조 쓰기(monotonic write), 쓰기 읽기(read-your-write) 및 읽기 뒤 쓰기(write-follows-read) 보장을 적용하도록 합니다. 이 설정은 단일 지역과 글로벌로 배포되는 애플리케이션에서 모두 가장 널리 사용되는 일관성 수준입니다.
- **일관된 접두사 일관성(consistent prefix)**: 반환되는 업데이트는 간격 없이 모든 업데이트의 일부 접두사를 포함합니다. 일관적인 접두사 일관성 수준은 읽기가 잘못된 순서의 쓰기를 보지 않도록 보장합니다. 즉 쓰기가 "A, B, C" 순서로 수행된 경우 클라이언트는 "A", "A, B" 혹은 "A, B, C"를 볼 수 있지만 "A, C" 혹은 "B, A, C"와 같이 순서가 잘못된 순열은 볼 수 없습니다.
- **최종 일관성(Eventual)**: 읽기에 대한 순서가 보장되지 않습니다. 더 이상 쓰기가 없으면 복제본이 결국 수렴됩니다. 최종 일관성은 클라이언트가 이전에 읽은 값보다 더 오래된 값을 읽을 수 있기 때문에 가장 약한 형태의 일관성입니다. 즉 애플리케이션의 순서가 보장되지 않아도 되는 경우에 적합합니다.



TASK 04. 리소스 정리

1. Azure 포털에서 [04_cosmosDbRg 리소스 그룹] 블레이드로 이동한 후 메뉴에서 [리소스 그룹 삭제]를 클릭합니다.



2. 리소스 그룹 삭제 확인 창에서 리소스 그룹 이름을 입력한 후 [삭제]를 클릭합니다.

