Optimize query and operation performance in Azure Cosmos DB for NoSQL - 10

Optimize indexes in Azure Cosmos DB for NoSQL

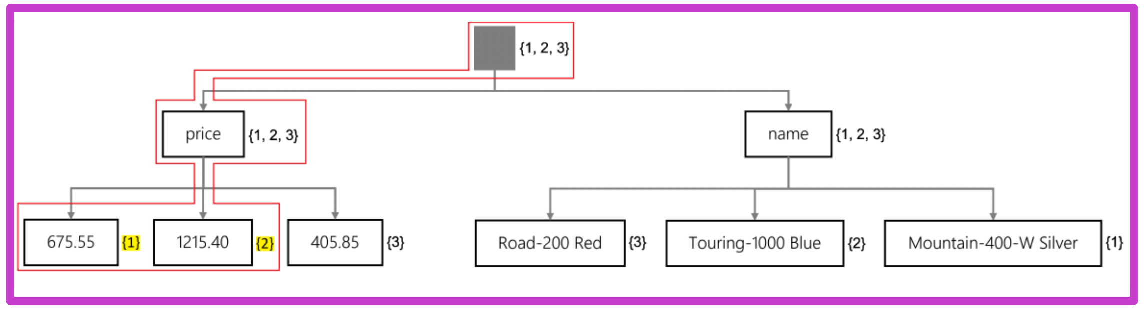
Index usage

The query engine will automatically try to use the most efficient method of evaluating filters, *index seek, index scan, full scan.*

Suppose the items in the *product* container are:

```
[
  { "id": "1", "name": "Mountain-400-W Silver", "price": 675.55 },
  { "id": "2", "name": "Touring-1000 Blue", "price": 1215.40 },
  { "id": "3", "name": "Road-200 Red", "price": 405.85 }
]
```

***product* inverted index tree:**



How will the index be used with the following queries?

```
SELECT *
FROM products p
WHERE p.name = 'Touring-1000 Blue'
```

```
SELECT *
FROM products p
WHERE p.name IN ('Road-200 Red', 'Mountain-400-W Silver')
```

```
SELECT *
FROM products p
WHERE p.price >= 500 AND p.price <= 1500
```

Review read-heavy index patterns

Read-centric workloads benefit from having an inverted index that includes as many fields as possible to maximize query performance and minimize request unit charges.

Consider this sample item in the *product* container. Consider that the applications querying items on this container never search or filter on the description or metadata properties.

```
{
    "id": "3324789",
    "name": "Road-200 Green",
    "price": 510.55,
    "description": "Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Cras faucibus, turpis ut pulvinar bibendum, sapien mauris fermentum magna, a
tincidunt magna diam tincidunt enim. Fusce convallis justo nulla, at
tristique diam tempus vel. Suspendisse potenti. Curabitur rhoncus neque vel
elit condimentum finibus. Nullam porta lorem vitae enim tincidunt elementum.
Vestibulum id felis sit amet neque commodo scelerisque. Suspendisse euismod
ex ut hendrerit eleifend. Quisque euismod consectetur vulputate.",
    "metadata": { "created_by": "sdfuouu",
                  "created_on": "2020-05-05T19:21:27.0000000Z",
                  "department": "cycling",
                  "sku": "RD200-G"
                }
}
```

Default index policy

```
{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [ { "path": "/*" } ],
  "excludedPaths": [ { "path": "/\"_etag\"/?" } ]
}
```

Proposed index policies

```
{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [ { "path": "/*" } ],
  "excludedPaths": [ { "path": "/description/?" },
                     { "path": "/metadata/*" } ]
}
```

```
{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [ { "path": "/name/?" },
                     { "path": "/price/?" } ],
  "excludedPaths": [ { "path": "/*" } ]
}
```

Review write-heavy index patterns

Insert or update operations also make the indexer update the inverted index with data from your newly created or updated item. The more properties you index the more RUs used by the indexer.

Consider this sample item in the *product* container.

```
{
  "id": "3324734",
  "name": "Road-200 Green",
  "internal": {
     "tracking": { "id": "eac06d51-2462-4bfb-8eb6-46281da16f8e" } },
  "inStock": true,
  "price": 1303.33,
  "description": "Consequat dolore commodo tempor pariatur consectetur
fugiat labore velit aliqua ut anim. Et anim eu ea reprehenderit sit ullamco
elit irure laborum sunt ea adipisicing eu qui. Officia commodo ad amet ea
consectetur ea est fugiat.",
  "warehouse": { "shelfLocations": [ 20, 37, 35, 27, 38 ] },
  "metadata": { "color": "brown",
                "manufacturer": "Fabrikam",
                "supportEmail": "support@fabrik.am",
                "created_by": "sdfuouu",
                "created_on": "2020-05-05T19:21:27.0000000Z",
                "department": "cycling",
                "sku": "RD200-B" },
  "tags": [ "pariatur", "et", "commodo", "ex", "tempor", "esse",
            "nisi", "ullamco", "Lorem", "ullamco", "ex", "ea",
            "laborum", "tempor", "consequat" ]
}
```

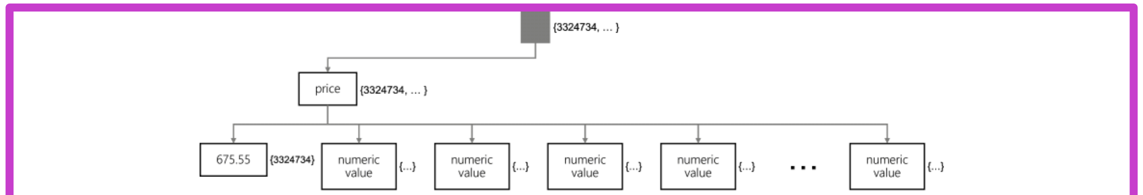
Assume the application only uses these two queries

```sql
SELECT *
FROM products p
WHERE p.price >= <numeric-value> AND p.price <= <numeric-value>
```

```sql
SELECT *
FROM products p
WHERE p.price = <numeric-value>
```

Proposed index policy

```
{
  "indexingMode": "consistent", "automatic": true,
  "includedPaths": [ { "path": "/price/?" } ],
  "excludedPaths": [ { "path": "/*" } ]
}
```



Measure index performance in Azure Cosmos DB for NoSQL

Enable indexing metrics

Azure Cosmos DB for NoSQL includes opt-in indexing metrics that illuminate how the current state of the index affects your query filters.

```
Container container = client.GetContainer("cosmicworks", "products");

string sql = "SELECT * FROM products p";

QueryDefinition query = new(sql);

//  PopulateIndexMetrics is disabled by default, enable it if troubleshooting query performance or are unsure how to modify your indexing policy.
QueryRequestOptions options = new()
{
   PopulateIndexMetrics = true
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOptions: options);

while(iterator.HasMoreResults)
{
    FeedResponse<Product> response = await iterator.ReadNextAsync();
    foreach(Product product in response)
    {
        Console.WriteLine($"[{product.id}]\t{product.name,35}\t{product.price,15:C}");
    }

    //  Do something with the metrics, in this example, we are sending it to the console output.
    Console.WriteLine(response.IndexMetrics);
}
```

Analyze indexing metrics results

Assume we are using the default index policy for the following queries.

Query 1

```
SELECT *
FROM products p
WHERE p.price > 500
```

Indexing metrics

```
Index Utilization Information
  Utilized Single Indexes
    Index Spec: /price/?
    Index Impact Score: High
    ---
  Potential Single Indexes
  Utilized Composite Indexes
  Potential Composite Indexes
```

Query 2

```
SELECT *
FROM products p
WHERE p.price > 500
    AND startsWith(p.name, 'Touring')
```

```
Index Utilization Information
  Utilized Single Indexes
    Index Spec: /price/?
    Index Impact Score: High
    ---
    Index Spec: /name/?
    Index Impact Score: High
    ---
  Potential Single Indexes
  Utilized Composite Indexes
  Potential Composite Indexes
```

Analyze indexing metrics results – Composite indexes

The indexing metrics could recommend we create a composite index.

Query 3

```
SELECT *
FROM products p
WHERE p.price > 500
    AND p.categoryName = 'Bikes, Touring Bikes'
```

Indexing metrics

```
Index Utilization Information
  Utilized Single Indexes
    Index Spec: /price/?
    Index Impact Score: High
    ---
    Index Spec: /categoryName/?
    Index Impact Score: High
    ---
  Potential Single Indexes
  Utilized Composite Indexes
  Potential Composite Indexes
    Index Spec: /categoryName ASC, /price ASC
    Index Impact Score: High
    ---
```

Add the potential composite index and run the query again.

```
{
  "indexingMode": "consistent", "automatic": true,
  "includedPaths": [ { "path": "/*" } ],
  "excludedPaths": [ { "path": "/\"_etag\"/?" } ],
  "compositeIndexes":
  [ [ { "path": "/categoryName", "order": "ascending" },
      { "path": "/price", "order": "ascending" }
  ] ]
}
```

```
Index Utilization Information
  Utilized Single Indexes
  Potential Single Indexes
  Utilized Composite Indexes
    Index Spec: /categoryName ASC, /price ASC
    Index Impact Score: High
    ---
  Potential Composite Indexes
```

Measure query cost

The QueryRequestOptions class is also helpful in measuring the cost of a query in RU/s.

```
Container container = client.GetContainer("cosmicworks", "products");
 string sql = "SELECT * FROM products p";
QueryDefinition query = new(sql);

// Set the MaxItemCount property of the QueryRequestOptions class to the number // of items you
would like to return in each result page.
QueryRequestOptions options = new()
{
   MaxItemCount = 25
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOptions:
options);

double totalRUs = 0;
while(iterator.HasMoreResults)
{
    FeedResponse<Product> response = await iterator.ReadNextAsync();
    foreach(Product product in response)
    {  // Do something with each product
    }
    //  Outputs the RU/s cost for returning every 25-item iteration.
    Console.WriteLine($"RU/s:\t\t{response.RequestCharge:0.00}");
    totalRUs += response.RequestCharge;
}

//  Returns the total RU/s cost of returning all items in the container..
Console.WriteLine($"Total RUs:\t{totalRUs:0.00}");
```

Sample console output

```
RUs: 2.82
RUs: 2.82
RUs: 2.83
RUs: 2.84
RUs: 2.25
Total RUs: 13.56
```

Measure point operation cost

You can also use the .NET SDK to measure the cost, in RU/s, of individual operations.

```
Container container = client.GetContainer("cosmicworks", "products");

Product item = new(
    $"{Guid.NewGuid()}",
    $"{Guid.NewGuid()}",
    "Road Bike",
    500,
    "rd-bk-500"
);

ItemResponse<Product> response = await container.CreateItemAsync<Product>(item);

Product createdItem = response.Resource;

Console.WriteLine($"RUs:\t{response.RequestCharge:0.00}");
```

Sample console output

```
RUs: 7.05
```

Implement integrated cache

Review workloads that benefit from the cache

Workloads that consistently perform the same point read and query operations are ideal to use with the integrated cache.

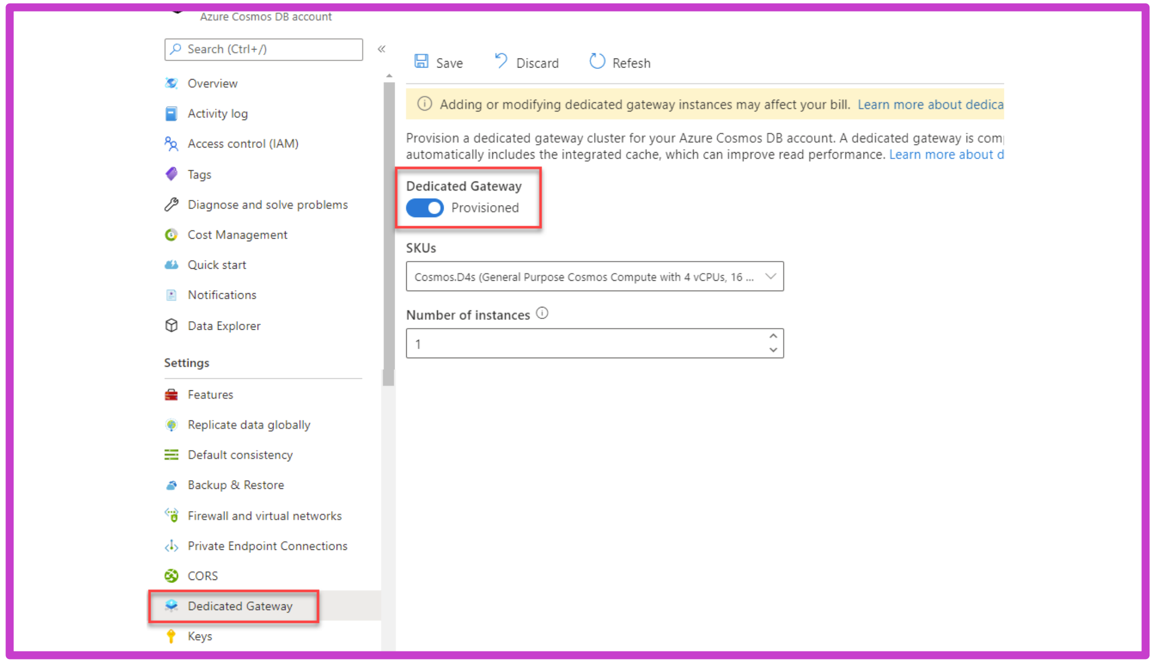
Workloads in Azure Cosmos DB that can benefit form the integrated cache:

- Workloads with far more queries than write operations
- Workloads that read large individual items multiple times
- Workloads that execute queries multiple times with a large amount of RU/s
- Workloads that have hot partition key[s] for read operations and queries

Enable integrated cache – Create a dedicated gateway

First step, Create a dedicated gateway in your Azure Cosmos DB for NoSQL account.

Create a dedicated gateway



Azure Cosmos DB account

Search (Ctrl+/)

- Overview
- Activity log
- Access control (IAM)
- Tags
- Diagnose and solve problems
- Cost Management
- Quick start
- Notifications
- Data Explorer

Settings

- Features
- Replicate data globally
- Default consistency
- Backup & Restore
- Firewall and virtual networks
- Private Endpoint Connections
- CORS
- Dedicated Gateway
- Keys

Save Discard Refresh

Adding or modifying dedicated gateway instances may affect your bill. Learn more about dedica

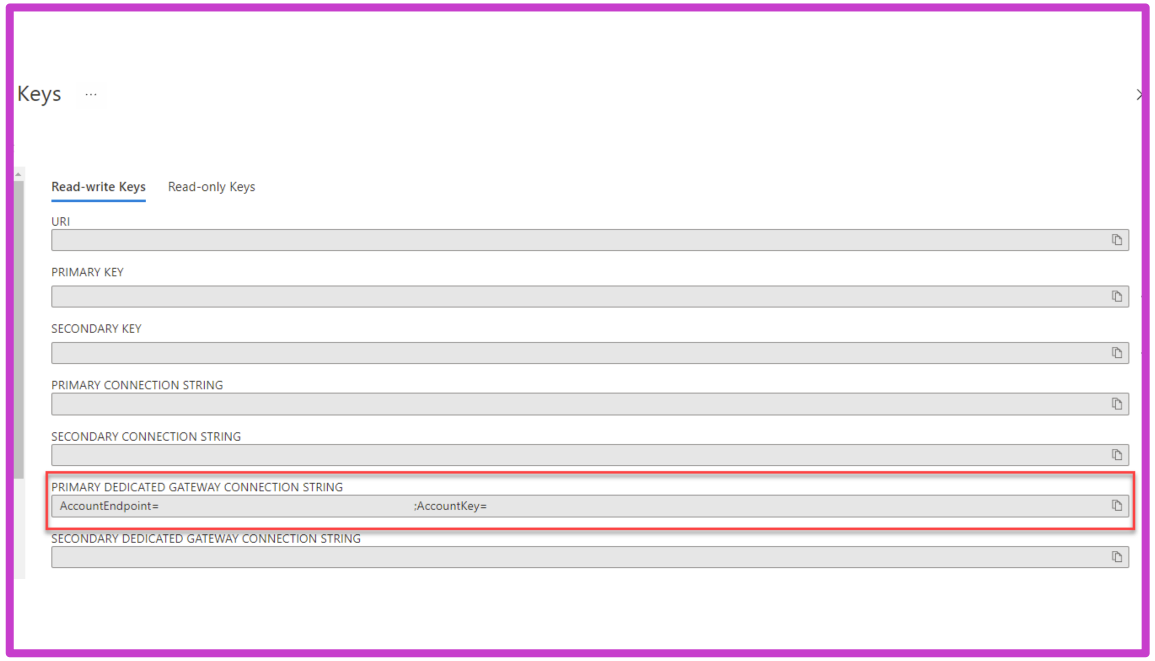
Provision a dedicated gateway cluster for your Azure Cosmos DB account. A dedicated gateway is com automatically includes the integrated cache, which can improve read performance. Learn more about d

Dedicated Gateway: Provisioned

SKUs: Cosmos.D4s (General Purpose Cosmos Compute with 4 vCPUs, 16 ...

Number of instances: 1

Get the connection string for the gateway



Keys

Read-write Keys | Read-only Keys

URI

PRIMARY KEY

SECONDARY KEY

PRIMARY CONNECTION STRING

SECONDARY CONNECTION STRING

PRIMARY DEDICATED GATEWAY CONNECTION STRING

AccountEndpoint= ;AccountKey=

SECONDARY DEDICATED GATEWAY CONNECTION STRING

Enable integrated cache – Update .NET SDK code

For the .NET SDK client to use the integrated cache you need the following changes:

The client uses the *dedicated gateway connection string* instead of the typical connection string

```
// For the dedicated gateway, the connection string is in the structure of <cosmos-account-name>.sqlx.cosmos.azure.com.
string connectionString = "AccountEndpoint=https://<cosmos-account-name>.sqlx.cosmos.azure.com/;AccountKey=<cosmos-key>;";
```

The client is configured to use *Gateway* mode instead of the default *Direct* connectivity mode

```
// Set the ConnectionMode property of the CosmosClientOptions class to ConnectionMode.Gateway.
CosmosClientOptions options = new()
{
    ConnectionMode = ConnectionMode.Gateway
};

CosmosClient client = new (connectionString, options);
```

The client's *consistency level* must be set to *session* or *eventual*

```
string sql = "SELECT * FROM products";
QueryDefinition query = new(sql);

// Set the ConsistencyLevel property of the QueryRequestOptions class to ConsistencyLevel.Session or ConsistencyLevel.Eventual.
QueryRequestOptions queryOptions = new()
{
   ConsistencyLevel = ConsistencyLevel.Eventual
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOptions: queryOptions);
```

Configure cache staleness

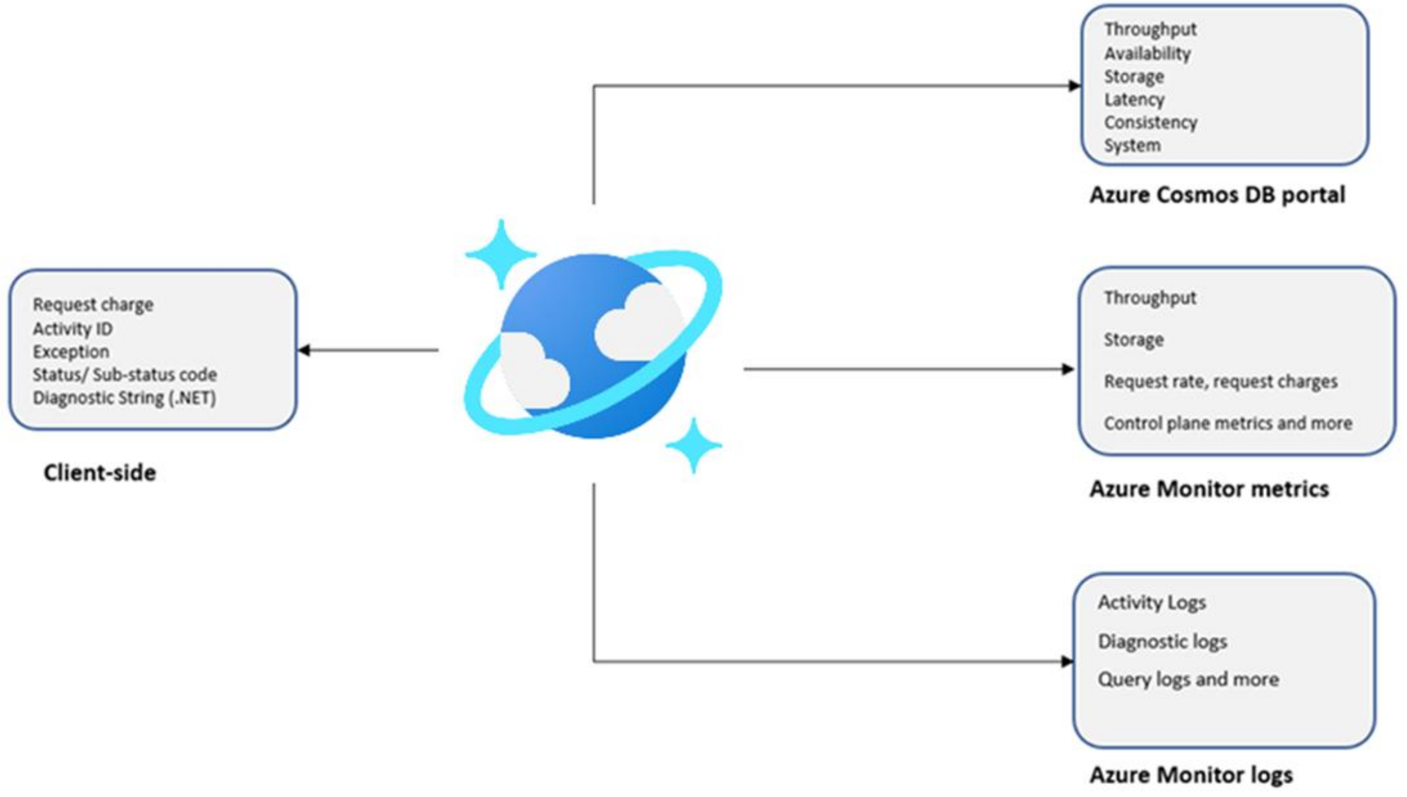
By default, the cache will keep data for five minutes. This staleness window can be configured using the *MaxIntegratedCacheStaleness* property in the SDK.

```
ItemRequestOptions operationOptions = new()
{
    ConsistencyLevel = ConsistencyLevel.Eventual,
    DedicatedGatewayRequestOptions = new()
    {
        MaxIntegratedCacheStaleness = TimeSpan.FromMinutes(15)
    }
};
```

```
QueryRequestOptions queryOptions = new()
{
    ConsistencyLevel = ConsistencyLevel.Eventual,
    DedicatedGatewayRequestOptions = new()
    {
        MaxIntegratedCacheStaleness = TimeSpan.FromSeconds(120)
    }
};
```

Monitor and troubleshoot an Azure Cosmos DB for NoSQL solution - 11

Measure performance in Azure Cosmos DB for NoSQL



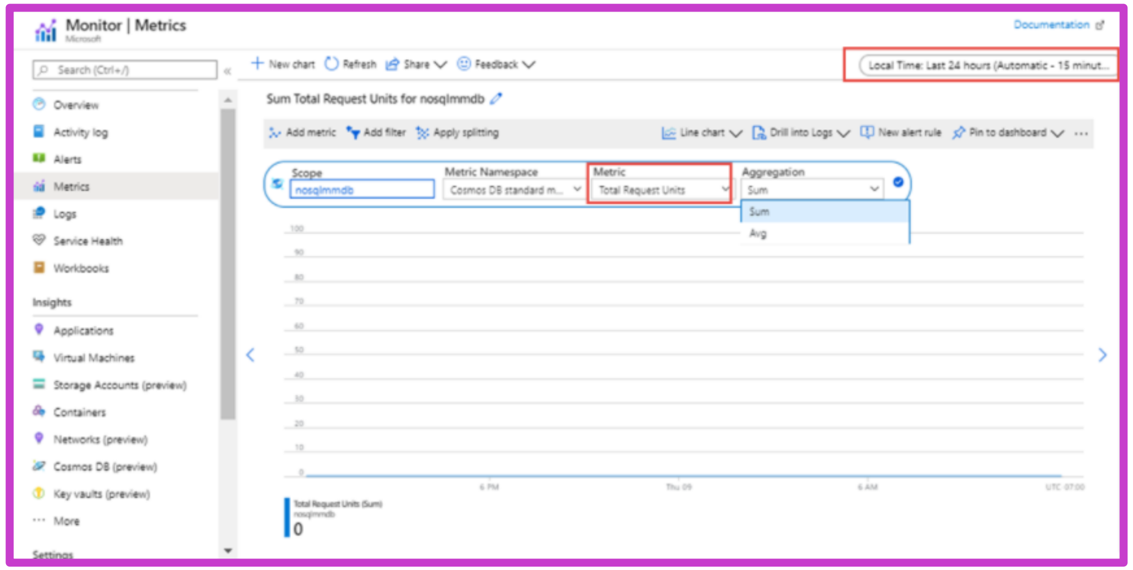
Understand Azure Monitor

- Azure Monitor is used to monitor the Azure resource availability, performance, and operations metrics.

Measure throughput

The Total Request Units metric can then be used to analyze those operations with the highest throughput.

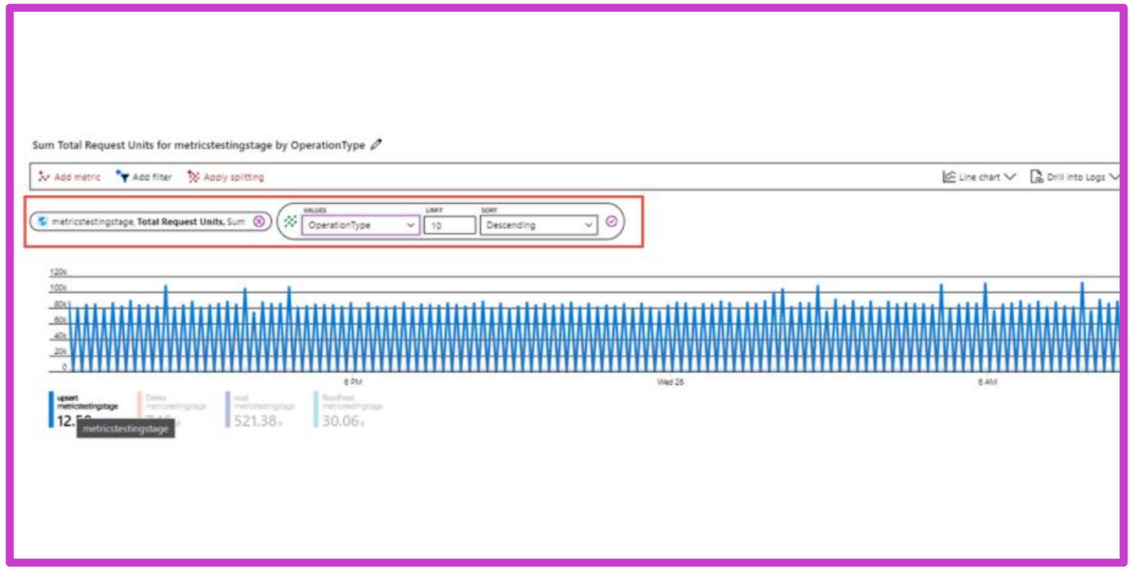
View the Total Request Unit metrics



Sum Total Request Units for nosqlmmdb

Scope: nosqlmmdb
Metric Namespace: Cosmos DB standard m...
Metric: Total Request Units
Aggregation: Sum

Filter the Total Request Units further



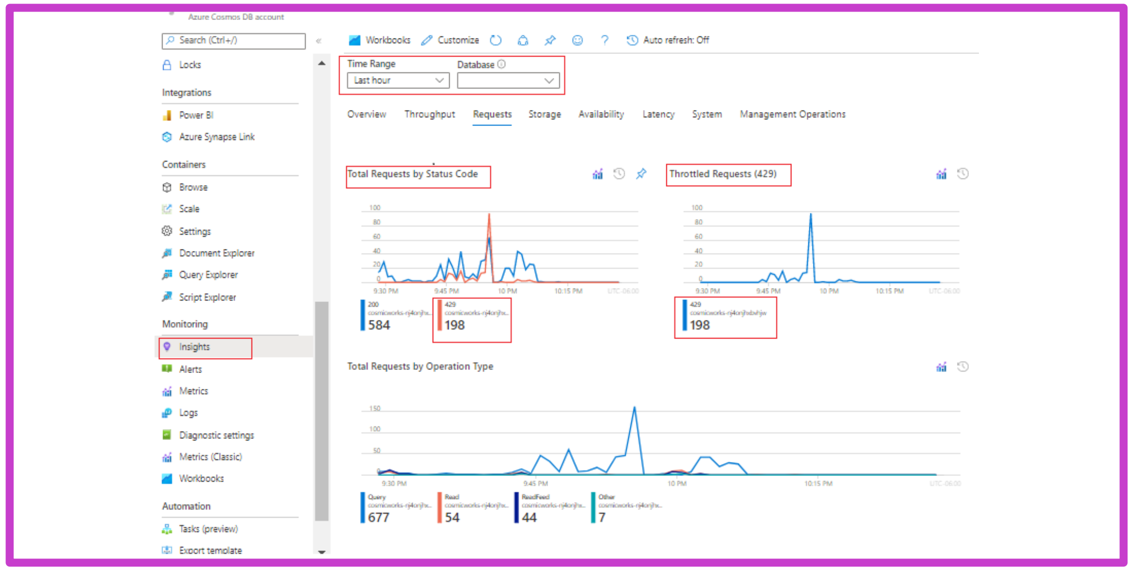
Sum Total Request Units for metricstestingstage by OperationType

metricstestingstage, Total Request Units, Sum
VALUES: OperationType
LIMIT: 10
SORT: Descending

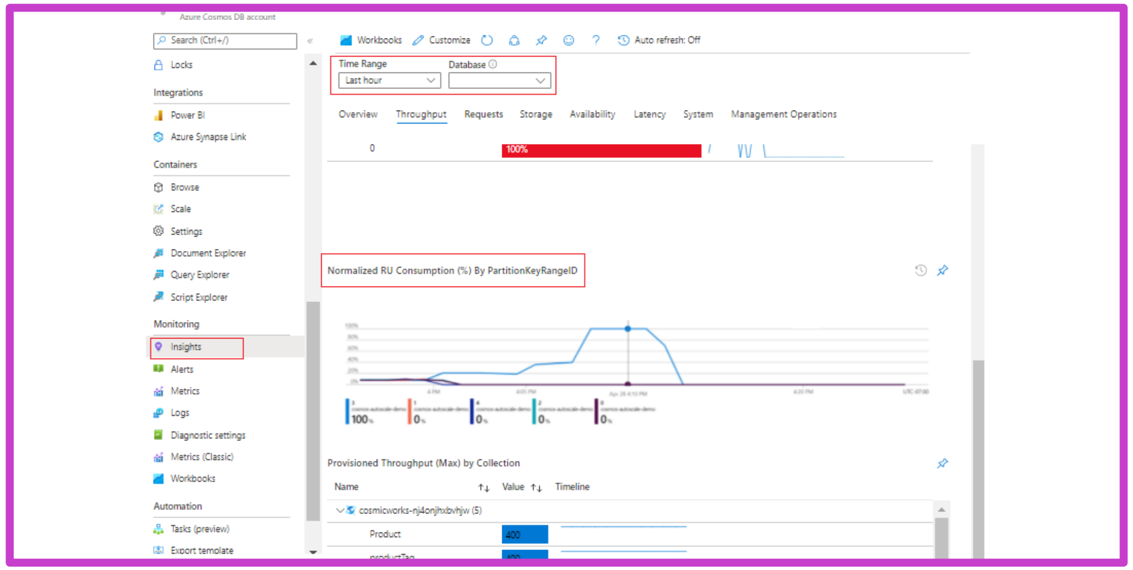
Observe rate-limiting events

The *429-status code* indicates that a *Request rate too large exception* has occurred. This exception means that Azure Cosmos DB requests are being rate limited. *

Review the Insights-Request charts for 429s



Review the Insights-Request charts for hot partitions

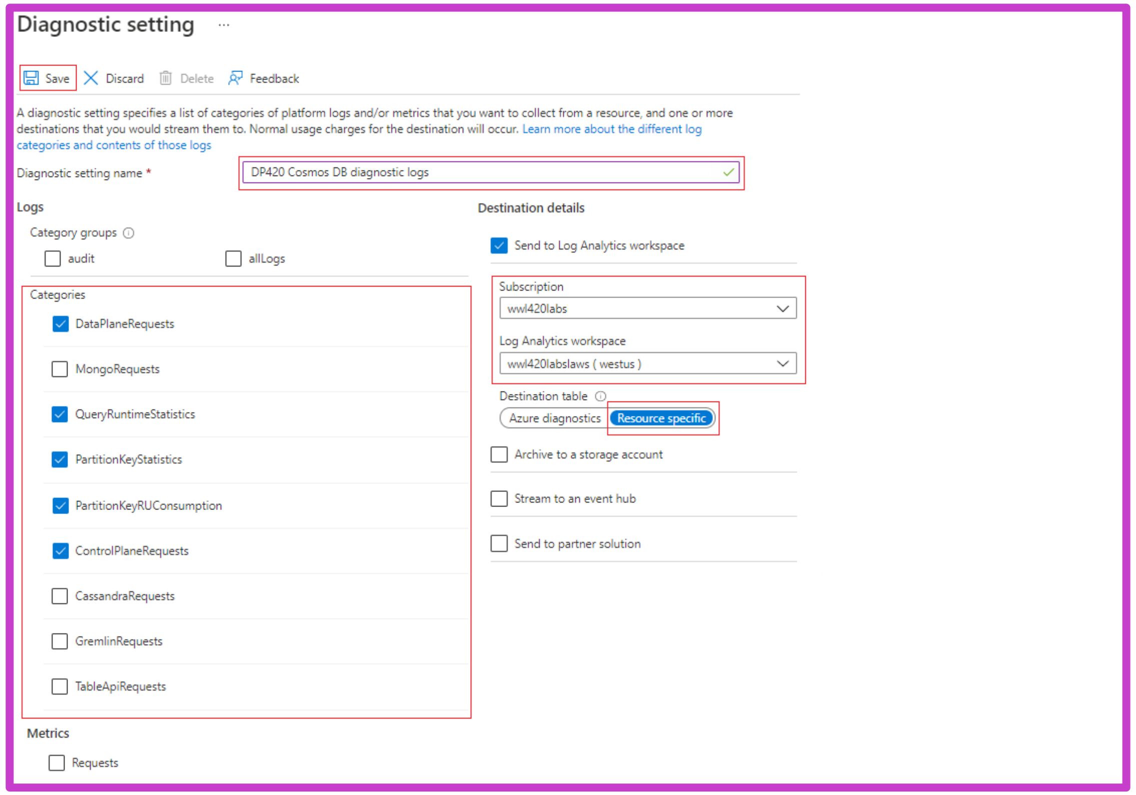


* Rate limiting exceptions can also be due to metadata request or transit service errors.

Query logs

Diagnostics settings are used to collect Azure Diagnostic Logs produced by Azure resources. These logs provide detailed resource operational data.

Create Azure Cosmos DB diagnostics settings


Diagnostic setting

Save | Discard | Delete | Feedback

A diagnostic setting specifies a list of categories of platform logs and/or metrics that you want to collect from a resource, and one or more destinations that you would stream them to. Normal usage charges for the destination will occur. Learn more about the different log categories and contents of those logs

Diagnostic setting name * DP420 Cosmos DB diagnostic logs

Logs

Category groups
- ☐ audit
- ☐ allLogs

Categories
- ✓ DataPlaneRequests
- ☐ MongoRequests
- ✓ QueryRuntimeStatistics
- ✓ PartitionKeyStatistics
- ✓ PartitionKeyRUConsumption
- ✓ ControlPlaneRequests
- ☐ CassandraRequests
- ☐ GremlinRequests
- ☐ TableApiRequests

Metrics
- ☐ Requests

Destination details
- ✓ Send to Log Analytics workspace

Subscription: wwl420labs

Log Analytics workspace: wwl420labslaws (westus)

Destination table: Azure diagnostics [✓ Resource specific]

- ☐ Archive to a storage account
- ☐ Stream to an event hub
- ☐ Send to partner solution

Troubleshoot issues with KQL diagnostics queries

```
// AzureDiagnostics queries
AzureDiagnostics
| where TimeGenerated >= ago(1h)
| where ResourceProvider=="MICROSOFT.DOCUMENTDB" and
Category=="DataPlaneRequests"
| summarize OperationCount = count(),
TotalRequestCharged=sum(todouble(requestCharge_s)) by
OperationName
| order by TotalRequestCharged desc
```

```
// Resource-specific Queries
CDBDataPlaneRequests
| where TimeGenerated >= ago(1h)
| summarize OperationCount = count(),
TotalRequestCharged=sum(todouble(RequestCharge)) by
OperationName
| order by TotalRequestCharged desc
```

Monitor responses and events in Azure Cosmos DB for NoSQL

Review common response codes

Azure Cosmos DB for NoSQL operations that create, query, or manage container documents, will return an HTTP operation status code.

Status Code	Name
200	OK
201	Created
204	No Content
304	Not Modified
400	Bad Request
403	Forbidden
404	Not Found
408	Request timeout
409	Conflict
413	Entity Too Large
429	Too many requests
500	Internal Server Error
503	Service Unavailable

Understand transient errors

We can identify and troubleshoot Azure Cosmos DB service unavailable exceptions when our request returns status code 503.

Required ports are blocked

Connection mode	Supported protocol	API/Service port
Gateway	HTTPS	NoSQL (443)
Direct	TCP	When using public/service endpoints: ports in the 10000 through 20000 range. When using private endpoints: ports in the 0 through 65535 range

Client-side transient connectivity issues

```
TransportException: A client transport error occurred: The request timed out while waiting for a server
response.
(Time: xxx, activity ID: xxx, error code: ReceiveTimeout [0x0010], base error: HRESULT 0x80131500
```

Service Outage

Check the *Azure status page* to see if there's an ongoing issue.

Review rate limiting errors

Requests return status code 429 for the exception request rate too large status code, indicating that your requests against Azure Cosmos DB are being rate-limited.

KQL query to determine which request types are causing 429 exceptions

```
// Resource-specific Queries
AzureDiagnostics
| where TimeGenerated >= ago(24h)
| where Category == "DataPlaneRequests"
| summarize throttledOperations = dcountif(activityId_g, statusCode_s == 429), totalOperations = dcount(activityId_g),
totalConsumedRUPerMinute = sum(todouble(requestCharge_s)) by databaseName_s, collectionName_s, OperationName,
requestResourceType_s, bin(TimeGenerated, 1min) | extend averageRUPerOperation = 1.0 * totalConsumedRUPerMinute /
totalOperations | extend fractionOf429s = 1.0 * throttledOperations / totalOperations
| order by fractionOf429s desc
```

Rate-limiting on metadata requests

Review occurrences of 429 exception in the Azure Cosmos DB *Insight* report *Metadata Requests That Exceeded Capacity (429s)* under the *System* tab.

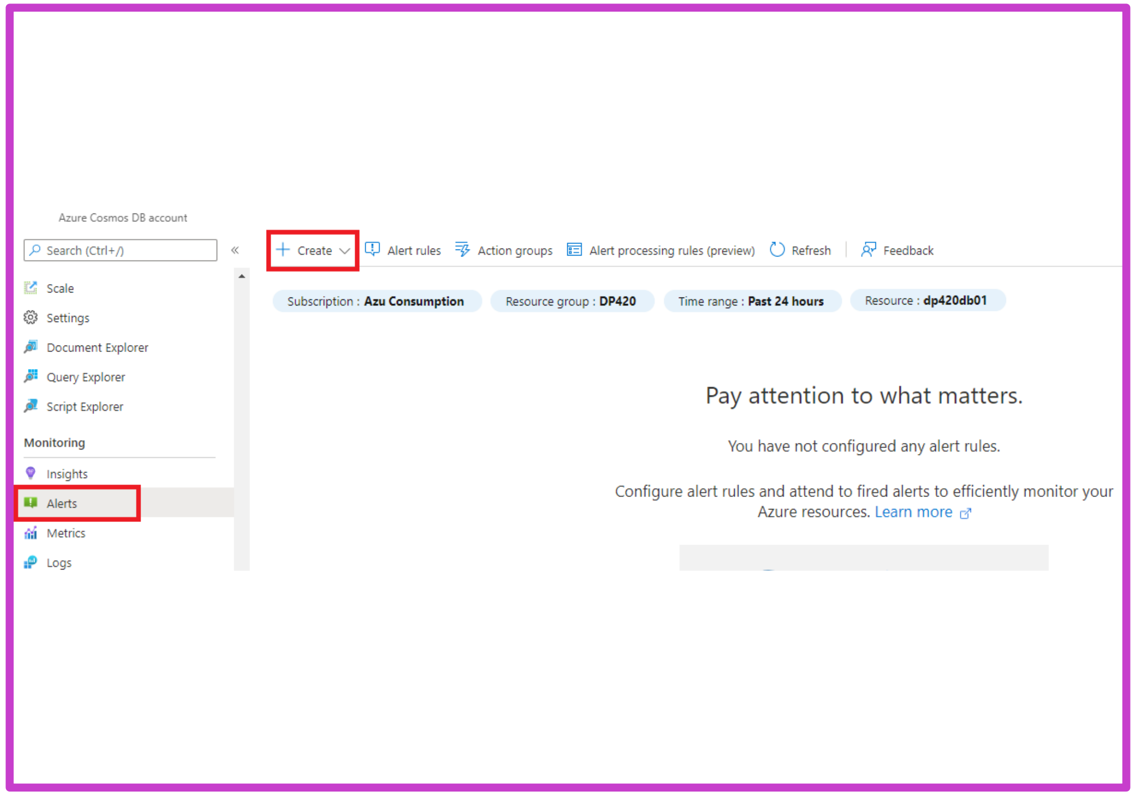
Rate-limiting due to transient service error

Retrying the request is the only recommended solution.

Configure Alerts

Azure Cosmos DB uses the Azure Monitor Service to set up and send alerts.

Create an alert



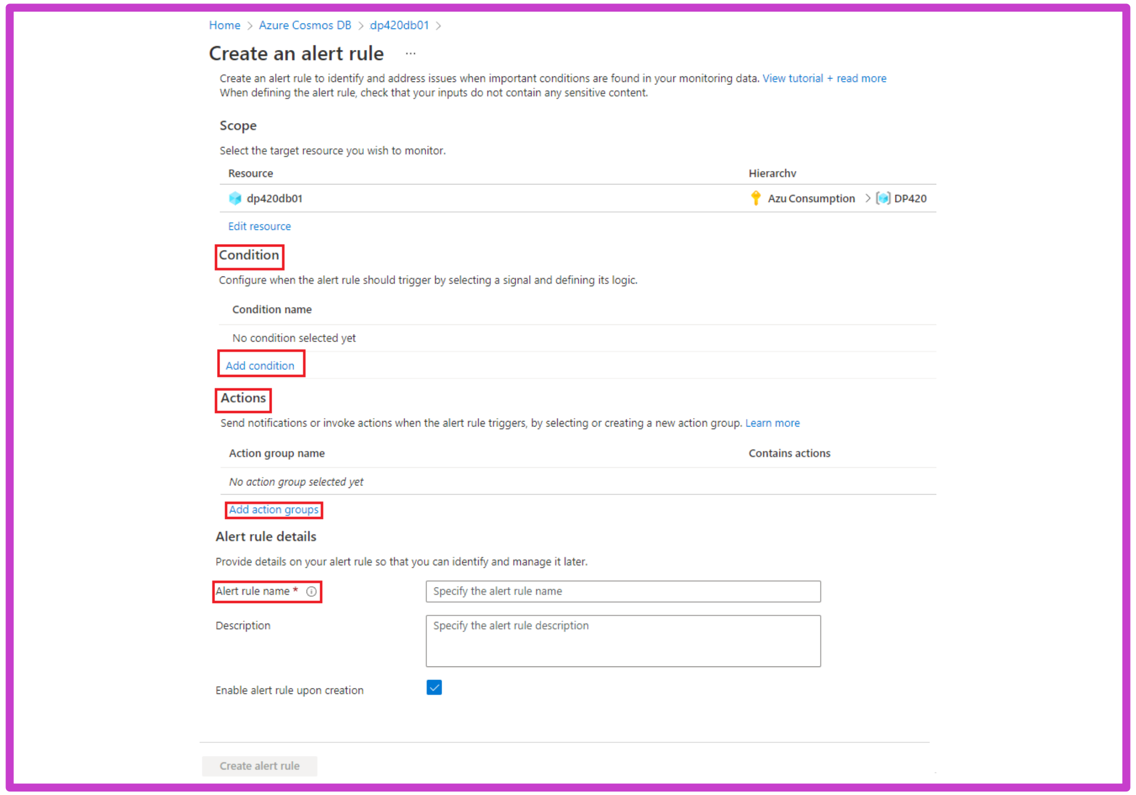
Subscription : **Azu Consumption** | Resource group : **DP420** | Time range : **Past 24 hours** | Resource : **dp420db01**

Pay attention to what matters.

You have not configured any alert rules.

Configure alert rules and attend to fired alerts to efficiently monitor your Azure resources. Learn more

Create alert rules



Create an alert rule

Create an alert rule to identify and address issues when important conditions are found in your monitoring data. View tutorial + read more
When defining the alert rule, check that your inputs do not contain any sensitive content.

Scope

Select the target resource you wish to monitor.

Resource	Hierarchv
dp420db01	Azu Consumption > DP420

Edit resource

Condition

Configure when the alert rule should trigger by selecting a signal and defining its logic.

Condition name
No condition selected yet

Add condition

Actions

Send notifications or invoke actions when the alert rule triggers, by selecting or creating a new action group. Learn more

Action group name	Contains actions
No action group selected yet	

Add action groups

Alert rule details

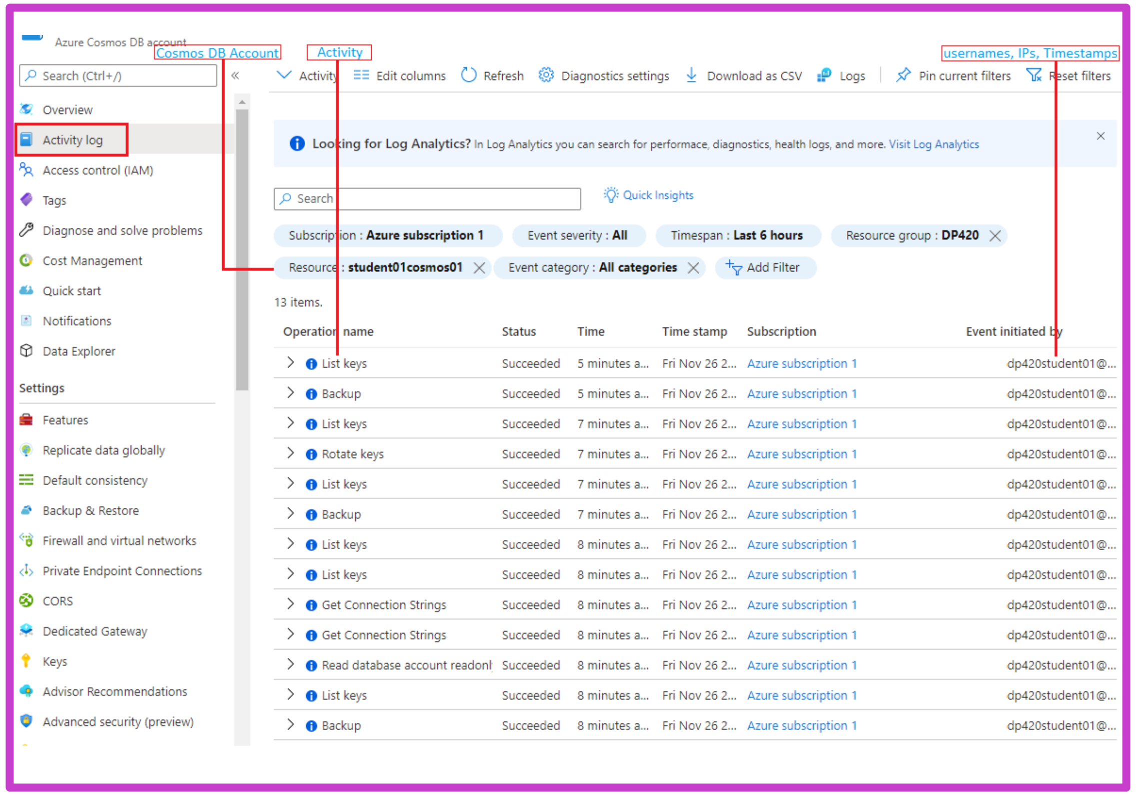
Provide details on your alert rule so that you can identify and manage it later.

Alert rule name *	Specify the alert rule name
Description	Specify the alert rule description
Enable alert rule upon creation	✓

Audit security

Azure Cosmos DB uses the Azure Monitor Service to set up and send alerts.

Activity Logs



Looking for Log Analytics? In Log Analytics you can search for performace, diagnostics, health logs, and more. Visit Log Analytics

Subscription : Azure subscription 1 | Event severity : All | Timespan : Last 6 hours | Resource group : DP420 | Resource : student01cosmos01 | Event category : All categories

13 items.

Operation name	Status	Time	Time stamp	Subscription	Event initiated by
List keys	Succeeded	5 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
Backup	Succeeded	5 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
List keys	Succeeded	7 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
Rotate keys	Succeeded	7 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
List keys	Succeeded	7 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
Backup	Succeeded	7 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
List keys	Succeeded	8 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
List keys	Succeeded	8 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
Get Connection Strings	Succeeded	8 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
Get Connection Strings	Succeeded	8 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
Read database account readonl	Succeeded	8 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
List keys	Succeeded	8 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...
Backup	Succeeded	8 minutes a...	Fri Nov 26 2...	Azure subscription 1	dp420student01@...

Azure Resource Logs

- Enable Azure resource logs for Cosmos DB.
- Enable the auditing control plane under Diagnostics settings.

Implementing backup and restore for Azure Cosmos DB for NoSQL

Evaluate periodic backup

Azure Cosmos DB takes automatic backups of your data at regular periodic intervals.

Backup Storage Redundancy	• Geo-redundant • Zone-redundant • Locally redundant
Change the default backup interval and retention period	• Backup Interval • Backup Retention • Backup storage redundancy
To request to restore a backup	Open a request ticket or call the Azure support team.
Consider restoring a backup when you …	• … deleted the entire Azure Cosmos DB account. • … deleted one or more Azure Cosmos DB databases. • … deleted one or more Azure Cosmos DB containers. • … deleted or modified the Azure Cosmos DB items within a container.
Costs of Extra backups	• Two backups included with the account for free. • Extra backups will be charged on a region-based backup-storing pricing.
Manage your own backups	• Azure Data Factory • Change feed

Configure continuous backup and recovery

When using the continuous backups mode, backups are continuously taken in every region where the Azure Cosmos DB account exists.

Backup Storage Redundancy	Change backup options	Continuous backup mode charges	Limitations – Not supported	Limitations
Locally redundant by default Zone-redundant when using Availability zones	Only option is to enable Continuous Backups Once set on a new or existing account can not be changed	Backup storage space. Restore cost. A separate charge will be added every time a restore is started.	Cosmos accounts using customer-managed keys. Multi-region write accounts. Accounts that create unique indexes after the container is created.	You can't restore an account into a region where the source account did not exist. The retention period is 30 days and can't be changed. Point in time restore always restores to a new Azure Cosmos DB account.

Perform a point-in-time recovery

Point-in-time recovery will allow you to choose any timestamp within the up to 30-days backup retention period and restore a combination of Azure DB containers, databases, or the accounts.

Restore Scenarios

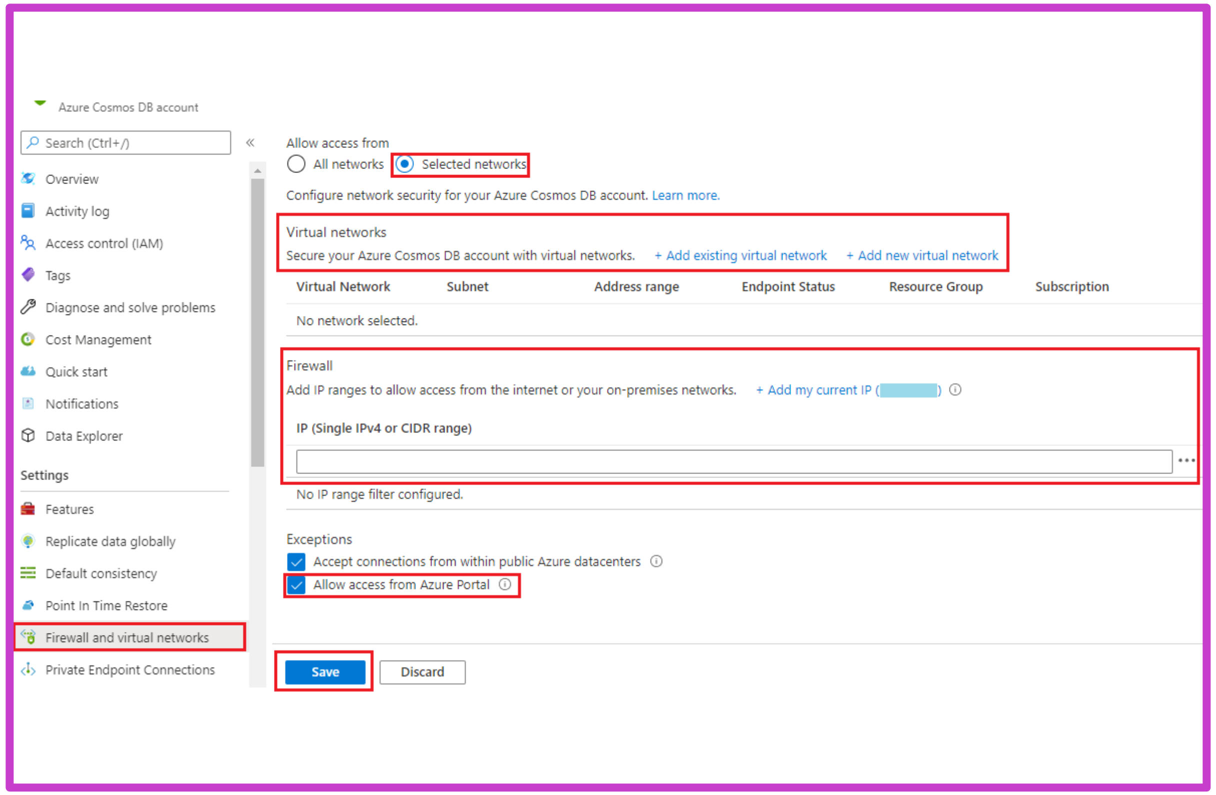
- Restore deleted account
- Restore data of an account in a particular region
- Recover from an accidental write or delete operation within a container with a known restore timestamp
- Restore an account to a previous point in time before the accidental delete of the database
- Restore an account to a previous point in time before the accidental delete or modification of the container properties

Implement security in Azure Cosmos DB for NoSQL

Implement network-level access control

Azure Cosmos DB supports IP-based access controls for inbound firewall support.

Configure an IP firewall by using the Azure portal



Azure Cosmos DB account

Search (Ctrl+/)

Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Cost Management
Quick start
Notifications
Data Explorer

Settings

Features
Replicate data globally
Default consistency
Point In Time Restore
Firewall and virtual networks
Private Endpoint Connections

Allow access from
All networks [✓ Selected networks]

Configure network security for your Azure Cosmos DB account. Learn more.

Virtual networks
Secure your Azure Cosmos DB account with virtual networks. + Add existing virtual network + Add new virtual network

Virtual Network	Subnet	Address range	Endpoint Status	Resource Group	Subscription
No network selected.					

Firewall
Add IP ranges to allow access from the internet or your on-premises networks. + Add my current IP ( )

IP (Single IPv4 or CIDR range)

No IP range filter configured.

Exceptions
✓ Accept connections from within public Azure datacenters
✓ Allow access from Azure Portal

Save Discard

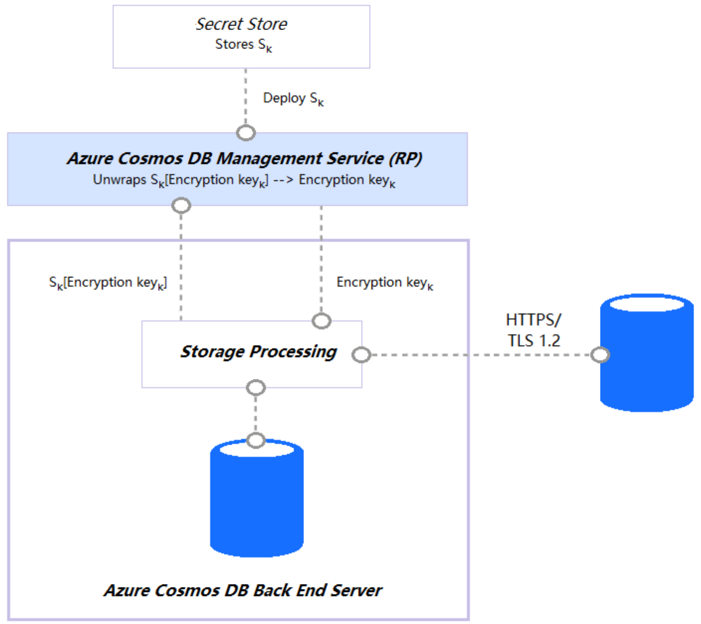
Troubleshoot issues with an IP access control policy

- Azure portal blocked
- SDK blocked
- Source IPs in blocked in requests
- Requests from a subnet with a service endpoint for Azure Cosmos DB enabled
- Private IP addresses in list of allowed addresses

Review data encryption options

Azure Cosmos DB now uses encryption at rest for all its databases, backups, and media. When Azure Cosmos DB data is in transit, or over the network, that data is also encrypted.

Azure Cosmos DB at rest and in transit encryption implementation

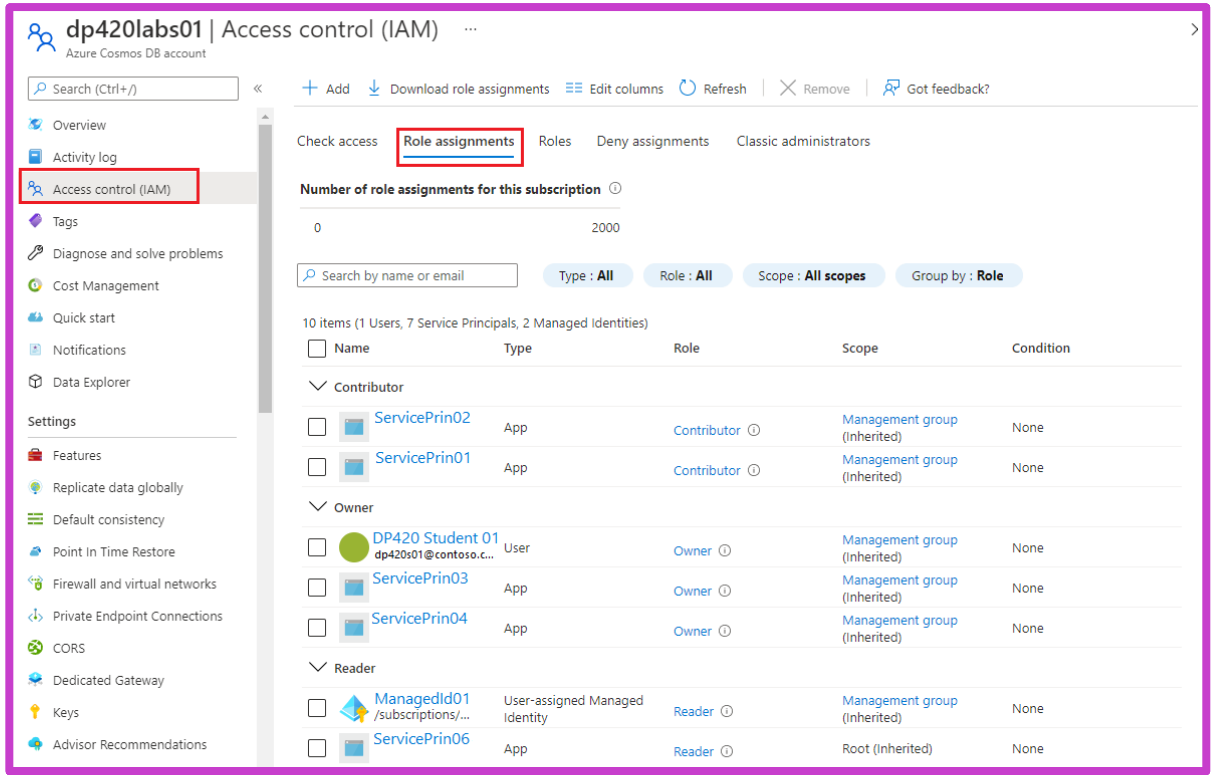


Secret Store
Stores S_K
Deploy S_K
Azure Cosmos DB Management Service (RP)
Unwraps S_K[Encryption key_K] --> Encryption key_K
S_K[Encryption key_K]
Encryption key_K
HTTPS/ TLS 1.2
Storage Processing
Azure Cosmos DB Back End Server

Use role-based access control (RBAC)

Azure role-based access control (RBAC) is provided in Azure Cosmos DB to do common management operations.

Identity and access management (IAM)



dp420labs01 | Access control (IAM) ...
Azure Cosmos DB account

Search (Ctrl+/)

Number of role assignments for this subscription

0 2000

Search by name or email

Type : **All** Role : **All** Scope : **All scopes** Group by : **Role**

10 items (1 Users, 7 Service Principals, 2 Managed Identities)

Name	Type	Role	Scope	Condition
Contributor				
ServicePrin02	App	Contributor	Management group (Inherited)	None
ServicePrin01	App	Contributor	Management group (Inherited)	None
Owner				
DP420 Student 01 dp420s01@contoso.c...	User	Owner	Management group (Inherited)	None
ServicePrin03	App	Owner	Management group (Inherited)	None
ServicePrin04	App	Owner	Management group (Inherited)	None
Reader				
ManagedId01 /subscriptions/...	User-assigned Managed Identity	Reader	Management group (Inherited)	None
ServicePrin06	App	Reader	Root (Inherited)	None

Other RBAC considerations

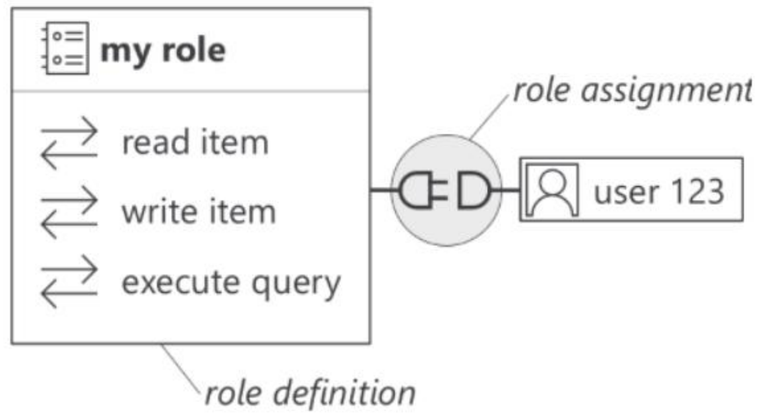
- Custom Controls
- Preventing changes from the Azure Cosmos DB SDKs

Access account resources using Microsoft Entra ID

Access account resources using Microsoft Entra allows you to authenticate your data requests with a Microsoft Entra identity.

Permission model

```
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/
containers/*
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/
containers/items/*
Microsoft.DocumentDB/databaseAccounts/readMetadata
```



Initialize the SDK with Azure AD

```
TokenCredential servicePrincipal = new
ClientSecretCredential(
    "<azure-ad-tenant-id>",
    "<client-application-id>",
    "<client-application-secret>“
);

CosmosClient client = new CosmosClient("<account-
endpoint>", servicePrincipal);
```

Manage an Azure Cosmos DB for NoSQL solution using DevOps practices - 12

Write management scripts for Azure Cosmos DB for NoSQL

Create resources

In this learning path, we will use Azure CLI to manage Azure Cosmos DB for NoSQL accounts.

Azure Cosmos DB account group commands

```
az cosmosdb create \
  --name '<account-name>’ \
  --resource-group '<resource-group>'
```

```
az cosmosdb create \
  --name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --default-consistency-level 'eventual’ \
  --enable-free-tier 'true'
```

```
az cosmosdb create \
  --name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --locations regionName='eastus'
```

```
az cosmosdb --help
```

```
az cosmosdb create --help
```

Azure Cosmos DB for NoSQL subgroup commands

```
az cosmosdb sql database create \
  --account-name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --name '<database-name>'
```

```
az cosmosdb sql container create \
  --account-name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --database-name '<database-name>’ \
  --name '<container-name>’ \
  --throughput '400’ \
  --partition-key-path '<partition-key-path-string>'
```

```
az cosmosdb sql --help
```

```
az cosmosdb sql database --help
```

```
az cosmosdb sql container --help
```

Manage index policies

When creating a container, you specify the indexing policy using CLI.

Let's assume that you have an indexing policy defined in a file named *policy.json*

```
{
   "indexingMode": "consistent",
   "automatic": true,
   "includedPaths": [ { "path": "/*" } ],
   "excludedPaths":
   [
        { "path": "/headquarters/*" },
        { "path": "/\"_etag\"/?" }
   ]
}
```

```
az cosmosdb sql container create \
  --account-name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --database-name '<database-name>’ \
  --name '<container-name>’ \
  --partition-key-path '<partition-key-path-string>’ \
  --idx '@.\policy.json’ \
  --throughput '400'
```

Raw JSON string

```
az cosmosdb sql container create \
  --account-name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --database-name '<database-name>’ \
  --name '<container-name>’ \
  --partition-key-path '<partition-key-path-string>’ \
  --idx
'{\"indexingMode\":\"consistent\",\"automatic\":true,\"includedPaths\":[{\"path\":\"/*\"}],\"excludedPaths\":[{\"path\":\"/headquarters/*\"},{
\"path\":\"/\\\"_etag\\\"/?\"}]}’ \
  --throughput '400'
```

Configure database or container-provisioned throughput

You can manage the provisioned throughput for both containers and databases using the CLI.

Update container throughput

```
az cosmosdb sql container throughput
update \
  --account-name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --database-name '<database-name>’ \
  --name '<container-name>’ \
  --throughput '1000'
```

Update database throughput

```
az cosmosdb sql database throughput
update \
  --account-name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --name '<database-name>’ \
  --throughput '4000'
```

Migrate between standard and autoscale throughput

Containers that manually provisioned throughput can be migrated to autoscale throughput.

Migrate throughput

```
az cosmosdb sql container throughput migrate \
  --account-name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --database-name '<database-name>’ \
  --name '<container-name>’ \
  --throughput-type 'autoscale'
```

```
az cosmosdb sql container throughput migrate \
  --account-name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --database-name '<database-name>’ \
  --name '<container-name>’ \
  --max-throughput '5000'
```

```
az cosmosdb sql container throughput migrate \
  --account-name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --database-name '<database-name>’ \
  --name '<container-name>’ \
  --throughput-type 'manual'
```

View the min throughput of an autoscale container

```
az cosmosdb sql container throughput show \
  --account-name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --database-name '<database-name>’ \
  --name '<container-name>’ \
  --query 'resource.minimumThroughput’ \
  --output 'tsv'
```

Configure failovers and failover priorities

Let’s assume that we have an Azure Cosmos DB account that we created in the *East US* region.

Add account regions

```
az cosmosdb update \
 --name '<account-name>’ \
 --resource-group '<resource-group>’ \
 --locations regionName='eastus' failoverPriority=0
isZoneRedundant=False \
 --locations regionName='westus2' failoverPriority=1
isZoneRedundant=False \
 --locations regionName='centralus' failoverPriority=2
isZoneRedundant=False
```

Remove account regions

```
az cosmosdb update \
  --name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --locations regionName='eastus' failoverPriority=0
isZoneRedundant=False \
  --locations regionName='westus2' failoverPriority=1
isZoneRedundant=False
```

Enable automatic failover

```
az cosmosdb update \
  --name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --enable-automatic-failover 'true'
```

Change failover priorities

```
az cosmosdb failover-priority-change \
  --name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --failover-policies 'eastus=0' 'centralus=1' 'westus2=2'
```

Enable multi-region write

```
az cosmosdb update \
  --name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --enable-multiple-write-locations 'true'
```

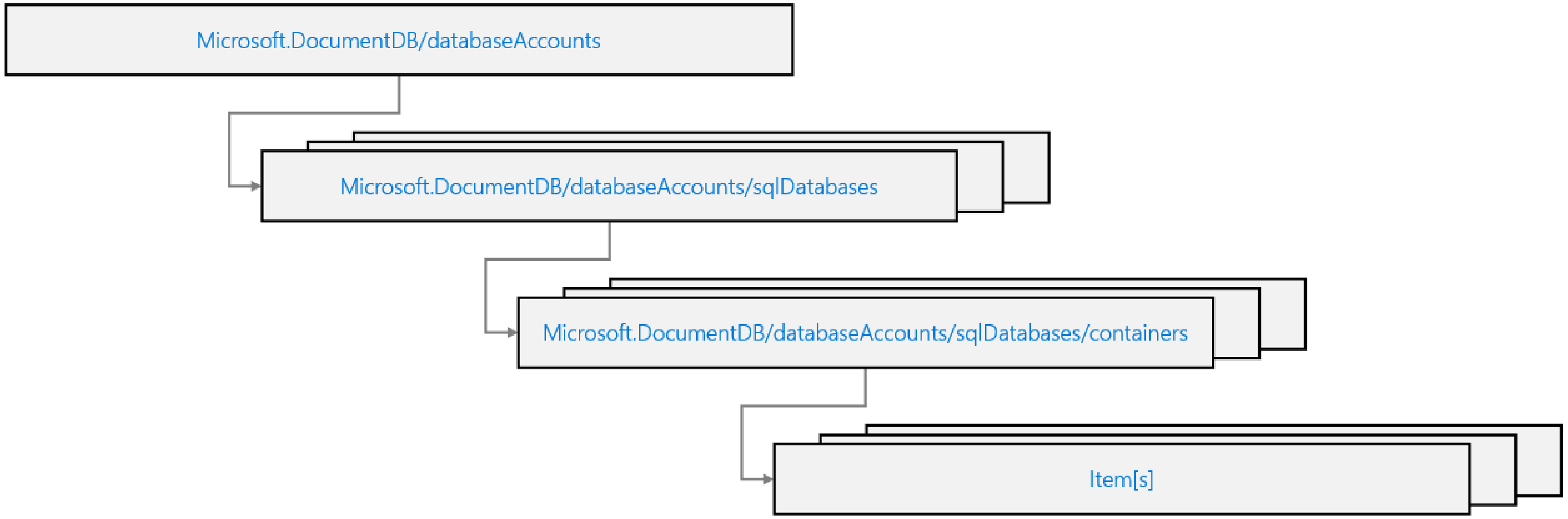
Initiate failovers

```
az cosmosdb failover-priority-change \
  --name '<account-name>’ \
  --resource-group '<resource-group>’ \
  --failover-policies 'westus2=0' 'eastus=1'
```

Create resource template for Azure Cosmos DB for NoSQL

Understand Azure Resource Manager resources

Each of the resources available for Azure Cosmos DB is listed under the *Microsoft.DocumentDB* resource provider.


Microsoft.DocumentDB/databaseAccounts
Microsoft.DocumentDB/databaseAccounts/sqlDatabases
Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers
Item[s]

Author Azure Resource Manager templates

There are three primary resources to define in a specific relationship order when authoring a template for an Azure Cosmos DB for NoSQL account.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "resources": [
      {
         "type": "Microsoft.DocumentDB/databaseAccounts",
         "apiVersion": "2024-04-15",
         "name": "[concat('csmsarm', uniqueString(resourceGroup().id))]",
         "location": "[resourceGroup().location]",
         "properties": {
               "databaseAccountOfferType": "Standard",
               "locations": [ { "locationName": "westus" } ]
          }
       },
       {
           "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases",
           "apiVersion": "2024-04-15",
           "name": "[concat('csmsarm', uniqueString(resourceGroup().id), '/cosmicworks')]",
           "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts', concat('csmsarm', uniqueString(resourceGroup().id)))]" ],
           "properties": { "resource": { "id": "cosmicworks" } }
       },
       {
           "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",
           "apiVersion": "2024-04-15",
           "name": "[concat('csmsarm', uniqueString(resourceGroup().id), '/cosmicworks/products')]",
           "dependsOn": [ "[resourceId('Microsoft.DocumentDB/databaseAccounts', concat('csmsarm', uniqueString(resourceGroup().id)))]",
                          "[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases', concat('csmsarm', uniqueString(resourceGroup().id)), 'cosmicworks')]"
                        ],
           "properties": { "options": { "autoscaleSettings": { “maxThroughput": 1000 } }, "resource": { "id": "products", "partitionKey": { "paths": [ "/categoryId" ] } } }
       }
   ]
}
```

Configure database or container-resources in Bicep

Each template resource uses the same resource type and version between both Azure Resource Manager and Bicep templates.

```
resource Account 'Microsoft.DocumentDB/databaseAccounts@2024-04-15' =
{
  name: 'csmsbicep${uniqueString(resourceGroup().id)}’
  location: resourceGroup().location
  properties: {
    databaseAccountOfferType: 'Standard’
    locations: [ { locationName: 'westus' } ]
  }
}

resource Database 'Microsoft.DocumentDB/databaseAccounts/sqlDatabases@2024-04-15’ =
{
  parent: Account name: 'cosmicworks’
  properties: {
    resource: { id: 'cosmicworks’ }
  }
}

resource Container 'Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers@2024-04-15’ =
{
  parent: Database name: 'customers’
  properties: {
    resource: {
      id: 'customers’
      partitionKey: { paths: [ '/regionId’ ] }
    }
  }
}
```

Deploy templates to a resource group

Now that the templates have been defined, use the Azure CLI to deploy either JSON or Bicep Azure Resource Manager templates.

Deploy Azure Resource Manager template to a resource group

```
az deployment group create \
  --resource-group '<resource-group>’ \
  --template-file '.\template.json'
```

```
az deployment group create \
--resource-group '<resource-group>’ \
--name '<deployment-name>’ \
--template-file '.\template.json'
```

```
az deployment group create \
  --resource-group '<resource-group>’ \
  --template-file '.\template.json’ \
  --parameters name='<value>'
```

```
az deployment group create \
  --resource-group '<resource-group>’ \
  --template-file '.\template.json’ \
  --parameters ‘@.\parameters.json'
```

Deploy Bicep template to a resource group

```
az deployment group create \
  --resource-group '<resource-group>’ \
  --template-file '.\template.bicep'
```

Manage index policies – JSON templates

Defining and deploying an indexing policy in JSON templates.

Defining an indexing policy in JSON templates

```
{
  "type": "Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers",
  "apiVersion": "2024-04-15",
  "name": "[concat('csmsarm', uniqueString(resourceGroup().id), '/cosmicworks/products')]",
  "dependsOn": [
                "[resourceId('Microsoft.DocumentDB/databaseAccounts', concat('csmsarm', uniqueString(resourceGroup().id)))]",
                "[resourceId('Microsoft.DocumentDB/databaseAccounts/sqlDatabases', concat('csmsarm', uniqueString(resourceGroup().id)), 'cosmicworks')]"
               ],
   "properties": {
                  "options": { "autoscaleSettings": { "maxThroughput": 1000 } },
                  "resource": {
                                "id": "products",
                                "partitionKey": { "paths": [ "/categoryId" ] },
                                "indexingPolicy": {
                                                    "indexingMode": "consistent",
                                                    "automatic": true,
                                                    "includedPaths": [ { "path": "/price/*" } ],
                                                    "excludedPaths": [ { "path": "/*" } ]
                                 }
                  }
   }
}
```

Deploying an indexing policy in JSON templates

```
az deployment group create \
  --resource-group '<resource-group>’ \
  --template-file '.\template.json’ \
  --name 'jsontemplatedeploy'
```

Manage index policies – Bicep templates

Defining and deploying an indexing policy in Bicep templates.

Defining an indexing policy in Bicep templates

```
resource Container 'Microsoft.DocumentDB/databaseAccounts/sqlDatabases/containers@2021-05-15' = {
    parent: Database
    name: 'customers’
    properties: {
        resource: {
                      id: 'customers’
                      partitionKey: { paths: [ '/regionId' ] }
                      indexingPolicy: {
                                         indexingMode: 'consistent’
                                         automatic: true includedPaths: [ { path: '/address/*' } ]
                                         excludedPaths: [ { path: '/*' } ]
                                       }
                      }
    }
}
```

Deploying an indexing policy in Bicep templates

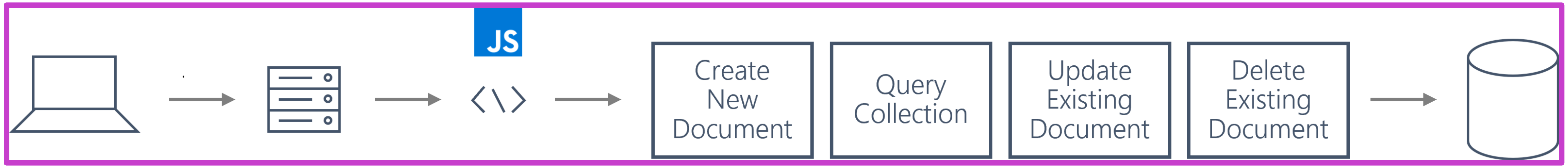
```
az deployment group create \
  --resource-group '<resource-group>’ \
  --template-file '.\template.bicep’ \
  --name 'biceptemplatedeploy'
```

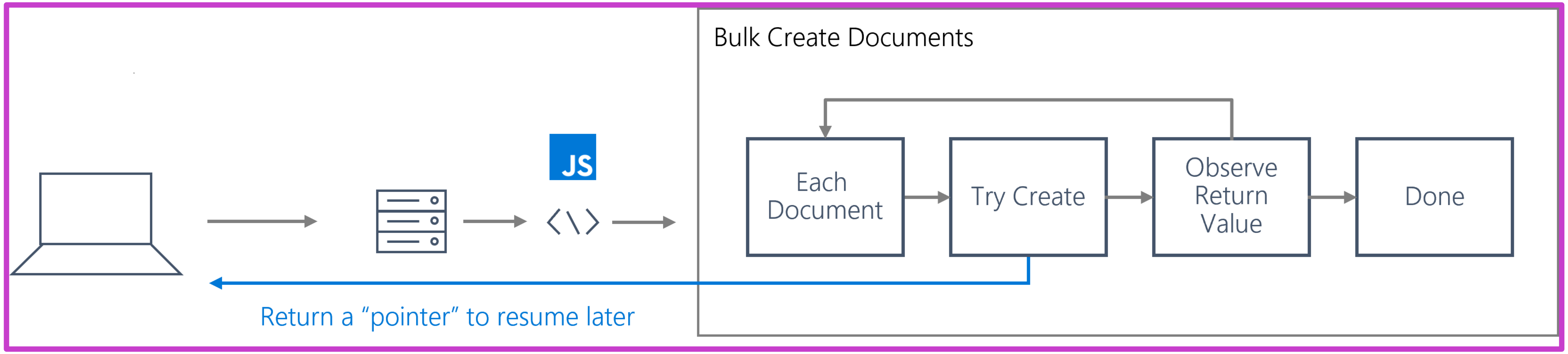
Create server-side programming constructs in Azure Cosmos DB for NoSQL-13

Build multi-item transactions with the Azure Cosmos DB for NoSQL

Understand transactions

In a database, a transaction is typically defined as a sequence of point operations grouped together into a single unit of work. It's expected that a transaction provides *ACID* guarantees.



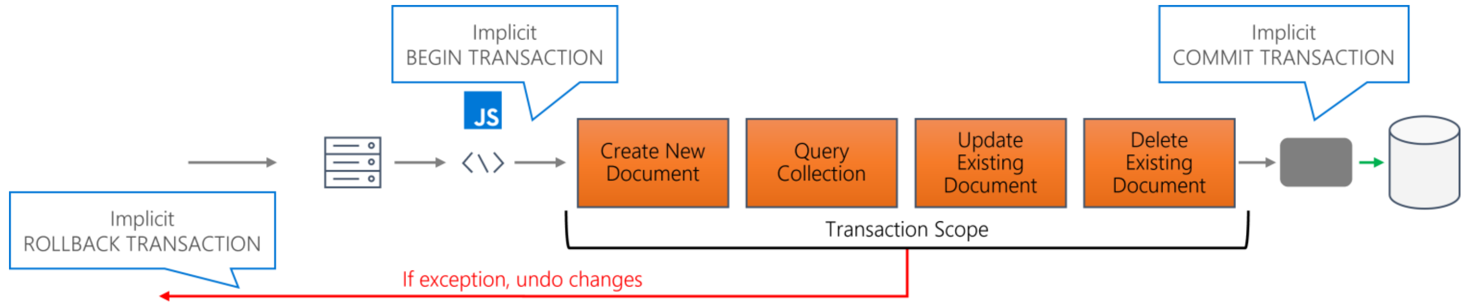


Author Stored procedures

Transactions are defined as *JavaScript* functions. The function is then executed when the stored procedure is invoked.

Azure Cosmos DB stored procedure to create an item

```
function createProduct(item)
{
    var context = getContext();
    var container = context.getCollection();
    var accepted = container.createDocument(
                                                    container.getSelfLink(),
                                                    item,
                                                    (
                                                        error,
                                                        newItem) => {
                                                                        if (error) throw error;
                                                                        context.getResponse().setBody(newItem)
                                                                    }
                                                    );
     if (!accepted) return;
}
```



Rollback transactions

Azure Cosmos DB's for NoSQL will roll back the entire transaction if a single exception is thrown from the stored procedure script.

Create stored procedures with the SDK

Creating a stored procedure using the .NET SDK requires the use of the *Scripts* property in the *Microsoft.Azure.Cosmos.Container* class.

```
//  First, define a string variable with the stored procedure’s JavaScript code.
string sproc =
@"function greet() {
    var context = getContext();
    var response = context.getResponse();
    response.setBody('Hello, Learn!’);
}";

//  Then, assign the string variable to the Body property of the StoreProcedureProperty class variable
StoredProcedureProperties properties = new()
{
    Id = "greet",
    Body = sproc
};

//  Finally, create the script by running Script.CreateStoredProcedureAsync with the stored procedure
properties defined
await container.Scripts.CreateStoredProcedureAsync(properties);
```

Expand query and transaction functionality in Azure Cosmos DB for NoSQL

Create User-defined functions (UDFs)

UDFs are used to extend the Azure Cosmos DB for NoSQL’s query language grammar and implement custom business logic and can only be called from inside queries.

Suppose you have the following item

```
{
  "name": "Black Bib Shorts (Small)",
  "price": 80.00
}
```

Create a UDF that calculates 15% tax

```
function addTax(preTax)
{
    return preTax * 1.15;
}
```

Run this query that returns the price and the tax

```
SELECT
    p.name,
    p.price,
    udf.addTax(p.price) AS priceWithTax
FROM products p
WHERE p.name = "Black Bib Shorts (Small)“
```

The query returns

```
[
  {
    "name": "Black Bib Shorts (Small)",
    "price": 80.00,
    "priceWithTax": 92.00
  }
]
```

Create User-defined functions (UDFs) with the SDK

The Scripts property in the *Microsoft.Azure.Cosmos.Container* class contains a *CreateUserDefinedFunctionAsync* method that is used to create a new UDF from code.

```
//  First, define a string variable with the UDF’s JavaScript code.
string udf =
@"function addTax(preTax)
{
    return preTax * 1.15;
}";

//  Then, assign the string variable to the Body property of the UserDefinedFunctionProperties class variable
UserDefinedFunctionProperties properties = new()
{
    Id = "addTax",
    Body = udf
};

//  Finally, create the script by running Script.CreateUserDefinedFunctionAsync with the UDF defined
await container.Scripts.CreateUserDefinedFunctionAsync(properties);
```

Add triggers to an operation – Pre-trigger

Pre-triggers are the core way that Azure Cosmos DB for NoSQL can inject business logic before an operations and cannot have any input parameters.

Suppose you want to insert the following item

```
{
  "id": "caab0e5e-c037-48a4-a760-140497d19452",
  "name": "Handlebar",
  "categoryId": "e89a34d2-47ee-4da8-bcf6-10f552604b79",
  "categoryName": "Accessories",
  "price": 50
}
```

The Item inserted will be

```
{
  "id": "caab0e5e-c037-48a4-a760-140497d19452",
  "name": "Handlebar",
  "categoryId": "e89a34d2-47ee-4da8-bcf6-10f552604b79",
  "categoryName": "Accessories",
  "price": 50,
  "label": "new"
}
```

Define the following pre-trigger

```
function addLabel(item)
{
    var context = getContext();
    var request = context.getRequest();
    var pendingItem = request.getBody();

    if (!('label' in pendingItem))
        pendingItem['label'] = 'new’;

    request.setBody(pendingItem);
}
```

Add triggers to an operation – Post-trigger

Post-triggers are the core way that Azure Cosmos DB for NoSQL can inject business logic after an operations completes and if needed, *can* have any input parameters.

Define the following post-trigger

```
function createView()
{
    var context = getContext();
    var container = context.getCollection();
    var response = context.getResponse();
    var createdItem = response.getBody();

    var viewItem = {
        sourceId: createdItem.id,
        categoryId: createdItem.categoryId,
        displayName: `${createdItem.name}
[${createdItem.categoryName}]`
    };

    var accepted = container.createDocument(
        container.getSelfLink(),
        viewItem, (
            error, newItem) => { if (error) throw error; }
        );

    if (!accepted) return;
}
```

When you insert the following item

```
{
  "id": "caab0e5e-c037-48a4-a760-140497d19452",
  "name": "Handlebar",
  "categoryId": "e89a34d2-47ee-4da8-bcf6-10f552604b79",
  "categoryName": "Accessories",
  "price": 50
}
```

The post-trigger will create this additional item

```
{
    "sourceId": "caab0e5e-c037-48a4-a760-140497d19452",
    "categoryId": "e89a34d2-47ee-4da8-bcf6-10f552604b79",
    "displayName": "Handlebar [Accessories]"
}
```

Create triggers with the SDK

The Scripts property in the *Microsoft.Azure.Cosmos.Container* class contains a *CreateTriggerAsync* method that is used to create a new pre/post trigger from code.

Define a Pre-trigger

```
string preTrigger =
@"function addLabel() {
    var context = getContext();
    var request = context.getRequest();

    var pendingItem = request.getBody();

    if (!('label' in pendingItem))
        pendingItem['label'] = 'new’;

    request.setBody(pendingItem);
}";

TriggerProperties properties = new()
{
    Id = "addLabel",
    Body = preTrigger,
    TriggerOperation = TriggerOperation.Create,
    TriggerType = TriggerType.Pre
};
await container.Scripts.CreateTriggerAsync(properties);
```

Define a Post-trigger

```
string postTrigger =
@"function createView() {
    var context = getContext();
    var container = context.getCollection();
    var response = context.getResponse();
    var createdItem = response.getBody();
    var viewItem = {
        sourceId: createdItem.id,
        categoryId: createdItem.categoryId,
        displayName: `${createdItem.name} [${createdItem.categoryName}]` };
    var accepted = container.createDocument(
        container.getSelfLink(),
        viewItem, (error, newItem) => { if (error) throw error; } );
    if (!accepted) return;
}";

TriggerProperties properties = new()
{
    Id = "createView",
    Body = postTrigger,
    TriggerOperation = TriggerOperation.Create,
    TriggerType = TriggerType.Post
};

await container.Scripts.CreateTriggerAsync(properties);
```

Use a trigger in an operation with the SDK

When the triggers have been defined and created within the container, you can use them in an operation on the same container.

Let’s suppose you want to create the following item

```json
{
  "id": "caab0e5e-c037-48a4-a760-140497d19452",
  "name": "Handlebar",
  "categoryId": "e89a34d2-47ee-4da8-bcf6-10f552604b79",
  "categoryName": "Accessories",
  "price": 50
}
```

Use the triggers when you create the item

```csharp
ItemRequestOptions options = new()
{
    PreTriggers = new List<string> { "addLabel" },
    PostTriggers = new List<string> { "createView" }
};

await container.CreateItemAsync(newItem, requestOptions:
options);
```

These triggers will create the following items

```json
[
  {
    "id": "caab0e5e-c037-48a4-a760-140497d19452",
    "name": "Handlebar",
    "categoryId": "e89a34d2-47ee-4da8-bcf6-10f552604b79",
    "categoryName": "Accessories",
    "price": 50,
    "label": "new"
  }

  {
      "sourceId": "caab0e5e-c037-48a4-a760-140497d19452",
      "categoryId": "e89a34d2-47ee-4da8-bcf6-10f552604b79",
      "displayName": "Handlebar [Accessories]"
  }
]
```