KodeKloud

© Copyright KodeKloud

Visit www.kodekloud.com to learn more.

Exploring Azure Cosmos DB

© Copyright KodeKloud

Introduction

01 Identify the key benefits of Azure Cosmos DB

02 Describe the elements in an Azure Cosmos DB account and how they are organized

03 Explain the different consistency levels and choose the correct one for your project

KodeKloud

04 Explore the APIs supported in Azure Cosmos DB and choose the appropriate API for your solution

05 Describe how request units impact costs

06 Create Azure Cosmos DB resources by using the Azure portal

© Copyright KodeKloud

Exploring Azure Cosmos DB

© Copyright KodeKloud

Azure Cosmos DB

© Copyright KodeKloud

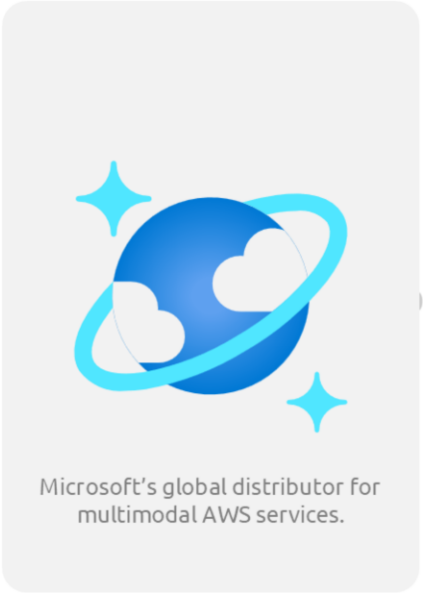
AI-powered web and mobile experiences
Develop innovative, data-driven applications that use the best of Azure AI. Add intelligence to your applications by integrating AI models through simple API calls

Simplified app development
Get fast, flexible app development with support for popular open-source databases like PostgreSQL, MongoDB, and Apache Cassandra.

Mission-critical performance at any scale
Build resilient,high-performance apps with SLA-backed <10 ms latency and 99.999% availability, global data replication, enterprise-grade security and continuous backup.

Fully managed and cost-effective
Keep costs and capacity in line with demand using serverless or autoscale options. Predictable vCore and node options meet your workload needs.

Azure Cosmos DB



© Copyright KodeKloud

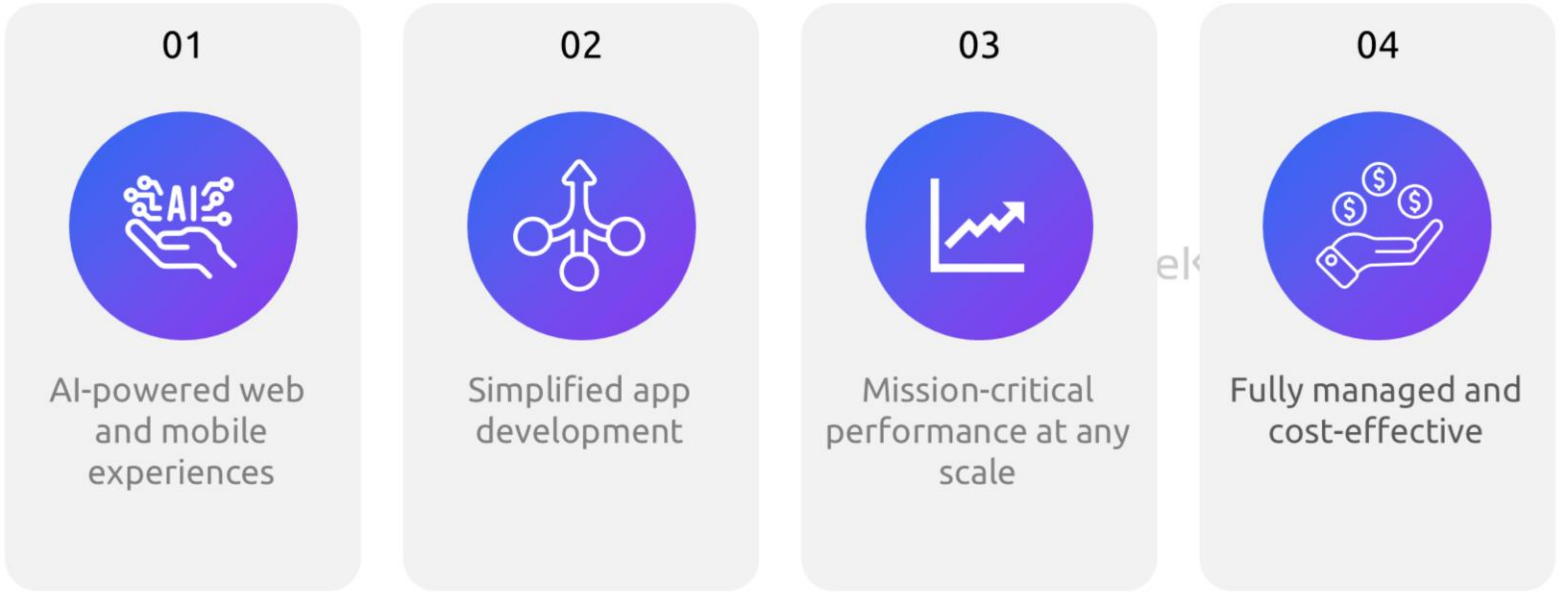
AI-powered web and mobile experiences
Develop innovative, data-driven applications that use the best of Azure AI. Add intelligence to your applications by integrating AI models through simple API calls

Simplified app development
Get fast, flexible app development with support for popular open-source databases like PostgreSQL, MongoDB, and Apache Cassandra.

Mission-critical performance at any scale
Build resilient,high-performance apps with SLA-backed <10 ms latency and 99.999% availability, global data replication, enterprise-grade security and continuous backup.

Fully managed and cost-effective
Keep costs and capacity in line with demand using serverless or autoscale options. Predictable vCore and node options meet your workload needs.

Azure Cosmos DB



© Copyright KodeKloud

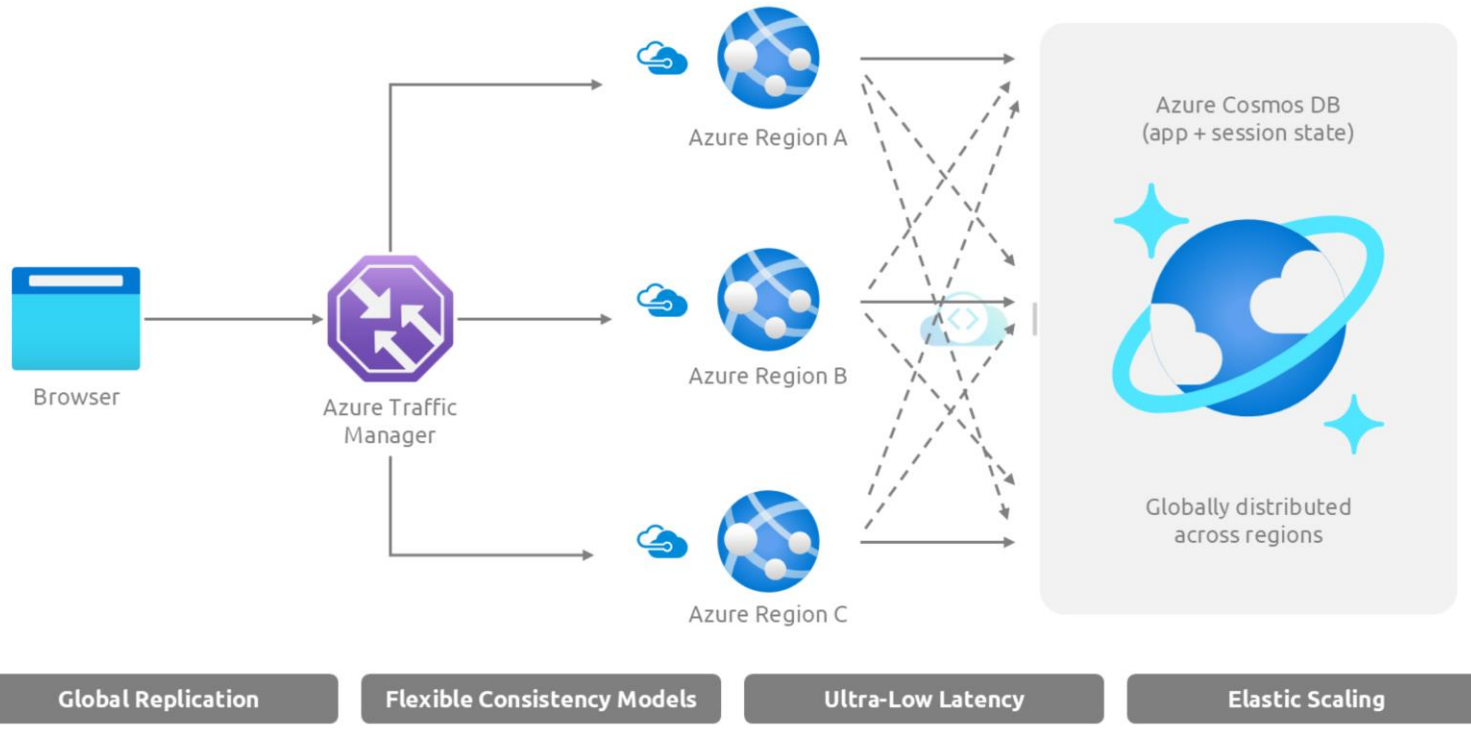
AI-powered web and mobile experiences
Develop innovative, data-driven applications that use the best of Azure AI. Add intelligence to your applications by integrating AI models through simple API calls

Simplified app development
Get fast, flexible app development with support for popular open-source databases like PostgreSQL, MongoDB, and Apache Cassandra.

Mission-critical performance at any scale
Build resilient,high-performance apps with SLA-backed <10 ms latency and 99.999% availability, global data replication, enterprise-grade security and continuous backup.

Fully managed and cost-effective
Keep costs and capacity in line with demand using serverless or autoscale options. Predictable vCore and node options meet your workload needs.

Azure Cosmos DB – Benefits



© Copyright KodeKloud

AI-powered web and mobile experiences
Develop innovative, data-driven applications that use the best of Azure AI. Add intelligence to your applications by integrating AI models through simple API calls

Simplified app development
Get fast, flexible app development with support for popular open-source databases like PostgreSQL, MongoDB, and Apache Cassandra.

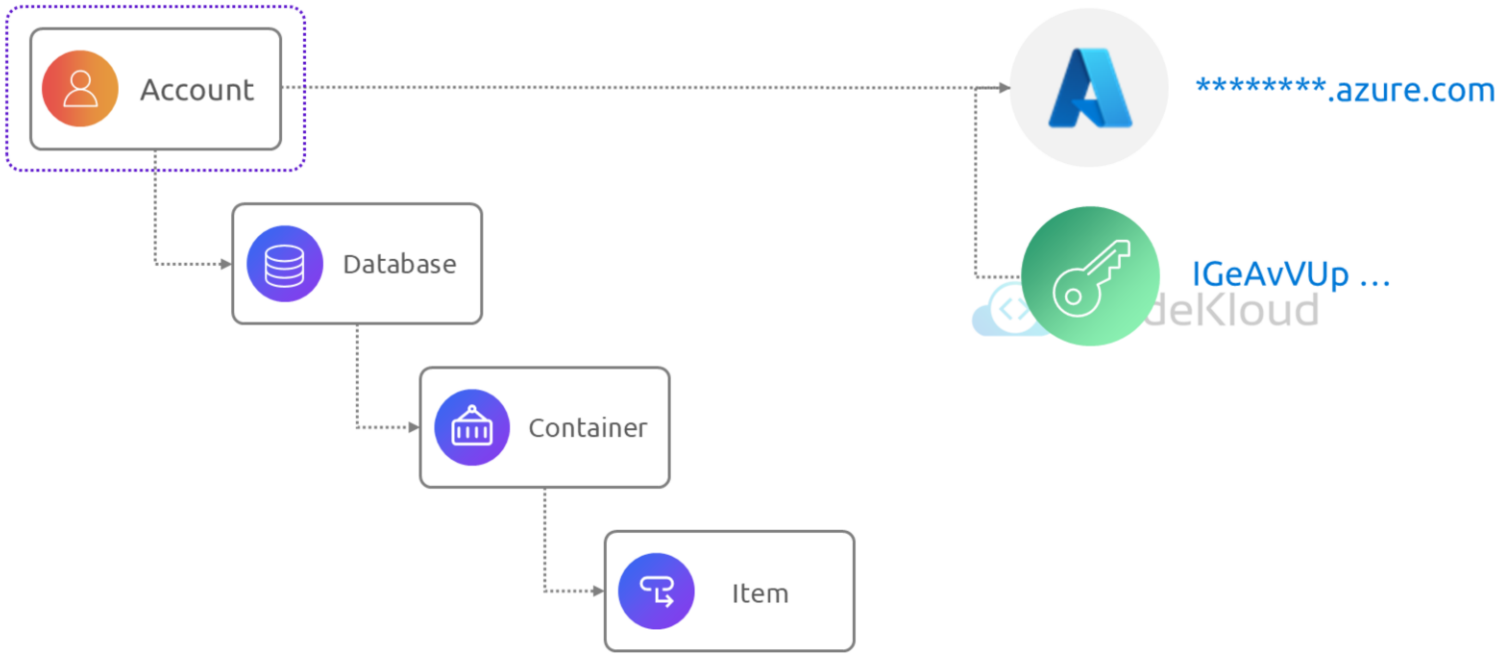
Mission-critical performance at any scale
Build resilient,high-performance apps with SLA-backed <10 ms latency and 99.999% availability, global data replication, enterprise-grade security and continuous backup.

Fully managed and cost-effective
Keep costs and capacity in line with demand using serverless or autoscale options. Predictable vCore and node options meet your workload needs.

Azure Cosmos DB Resource Hierarchy

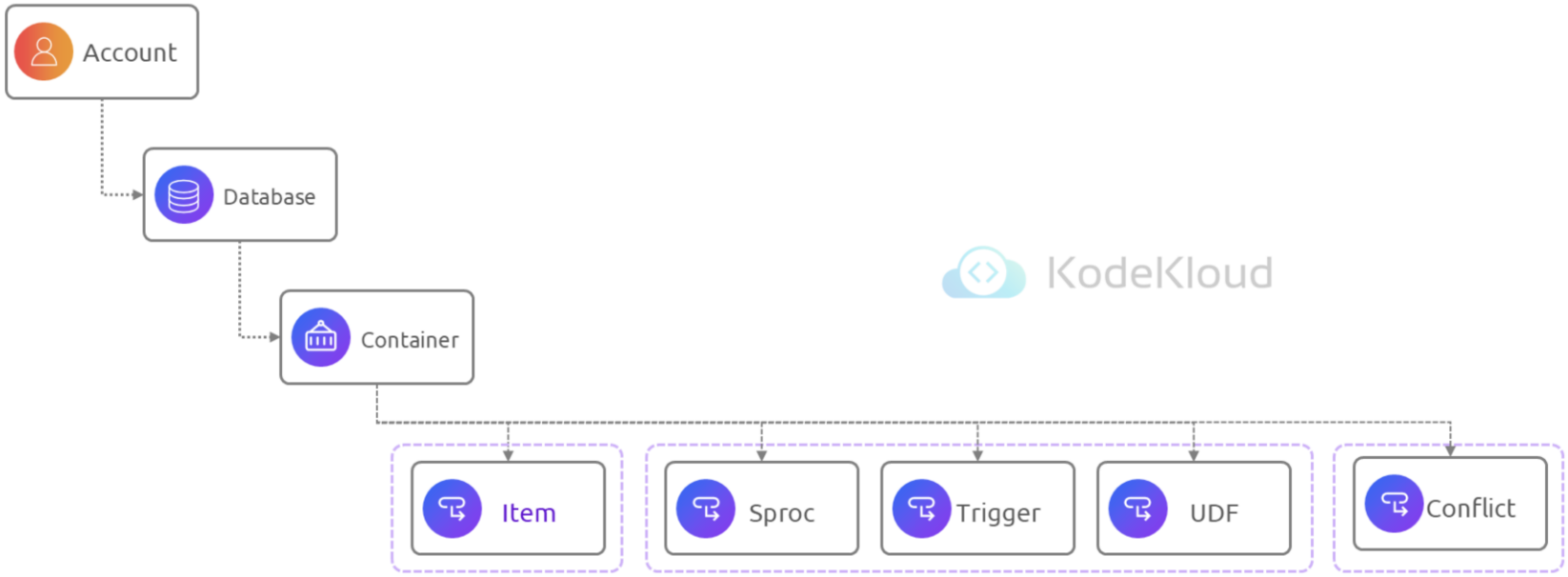
© Copyright KodeKloud

Resource Model – Account URI and Credentials



© Copyright KodeKloud

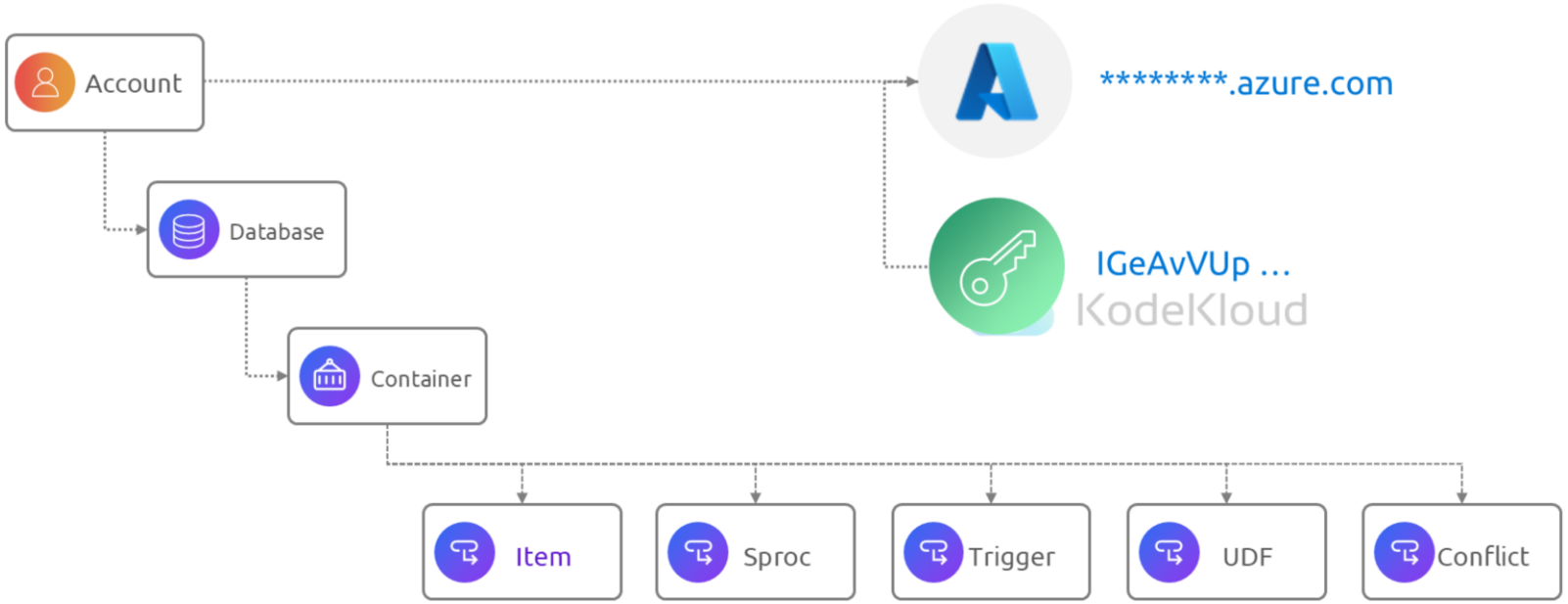
Resource Model – Database Resources



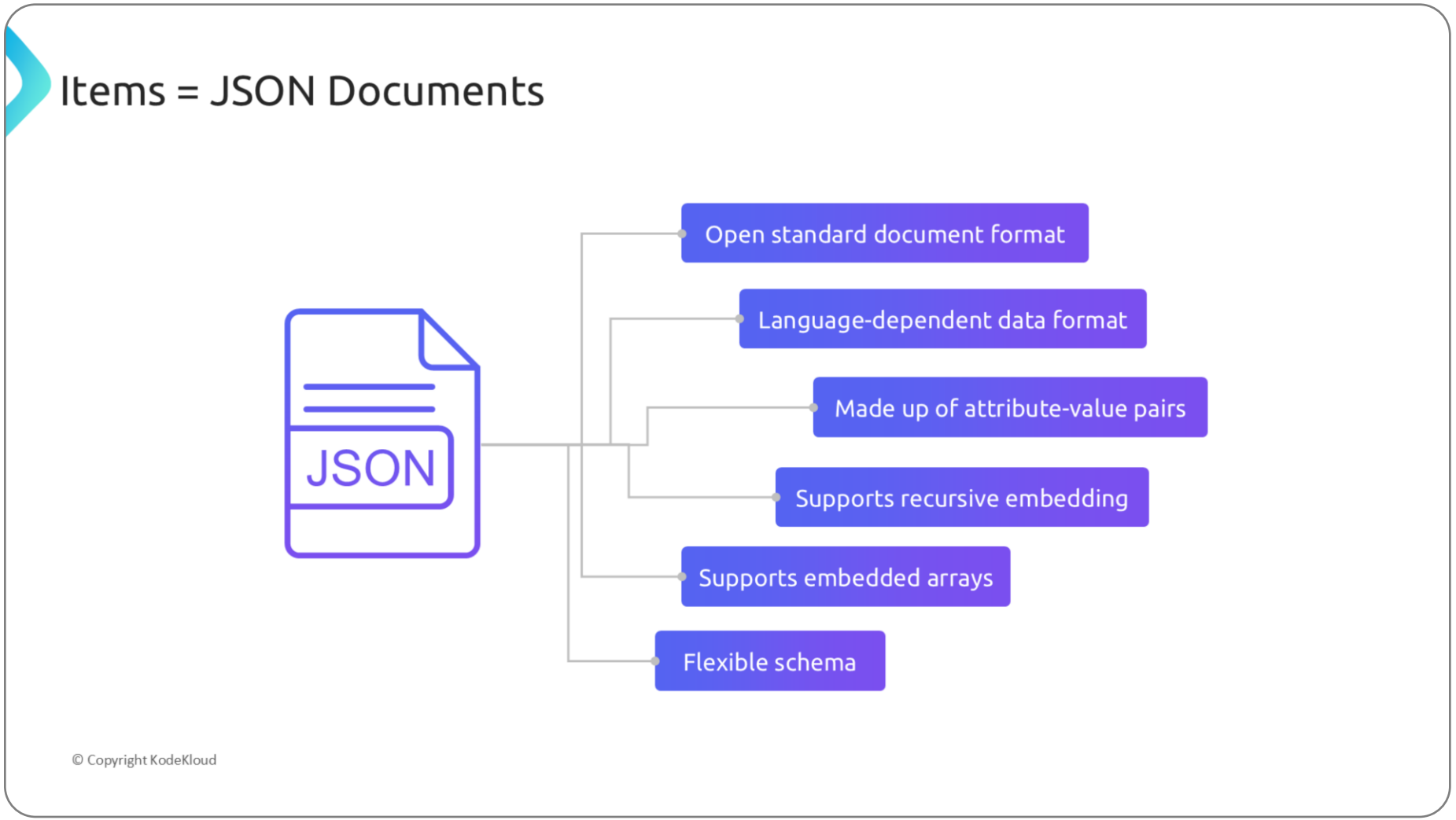
KodeKloud

© Copyright KodeKloud

Resource Hierarchy



© Copyright KodeKloud



Items = JSON Documents

Open standard document format
Language-independent data format
Made up of attribute–value pairs
Supports recursive embedding
Supports embedded arrays

Flexible schema

Up to 2MB for NoSQL API
Up to 16MB for MongoDB API

Items = JSON Documents

Terminal

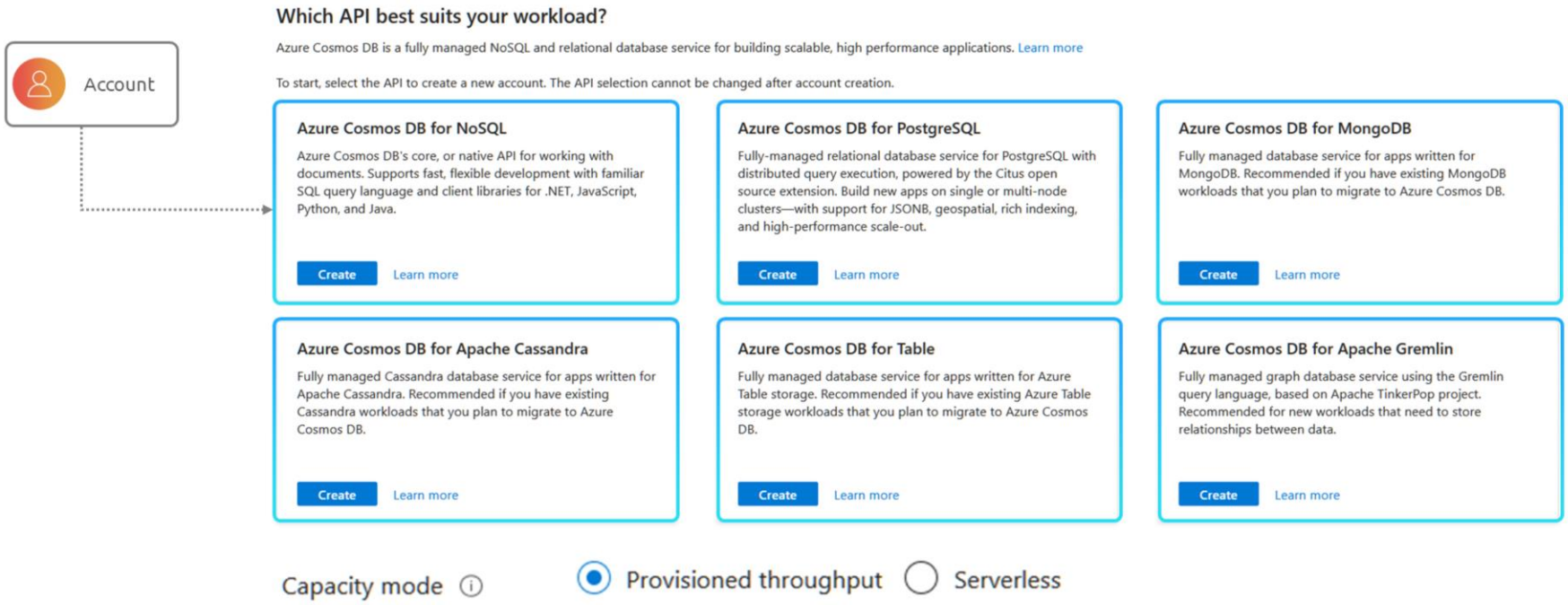
```
"InvoiceID": "IN12341287",
"TotalItems": 9,
"TotalValue": 52.15,
"Customer": {
        "CSID": 112423532,
        "FullnName": "Fred Flaire"
},
"Lines": [
  {
        "ProductCode": 63137,
        "Description": "Formal Work Pants (M)",
        "Quantity": 1,
        "Size": 32,
        "Color": "Black"
  },
  {
        "ProductCode": 63137,
        "Description": "Kitkat Jumbo",
        "Quantity": 8,
        "Size": 2.34,
        "Pack": 6,
        "Color": "Bars"
  }
 ]
}
```

© Copyright KodeKloud

Provisioning Azure Cosmos DB

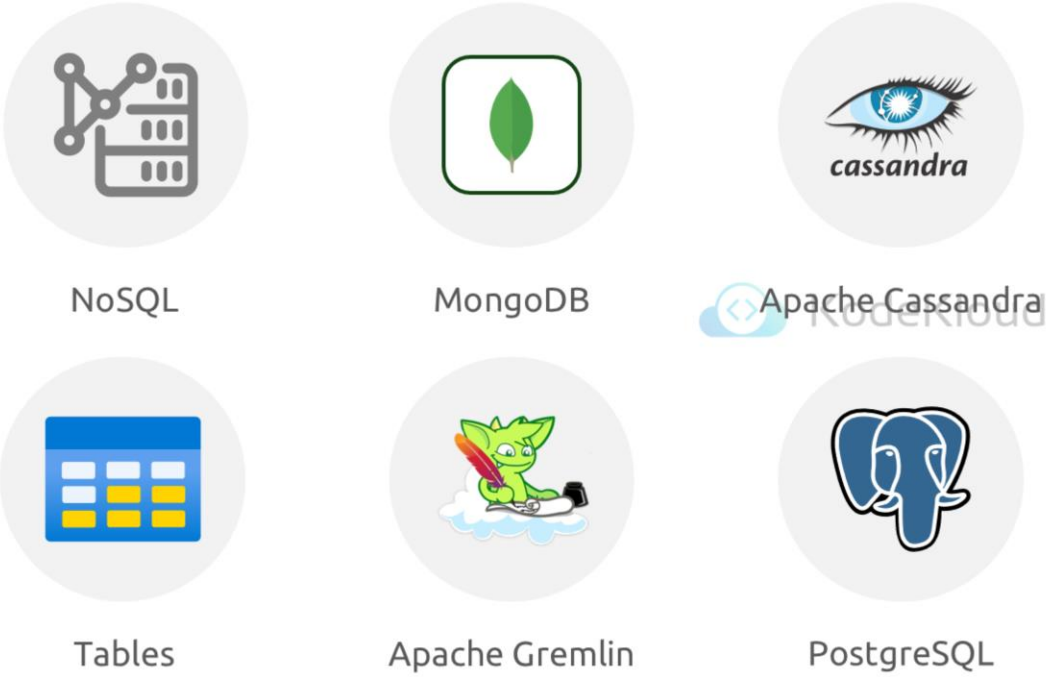
© Copyright KodeKloud

Resource Model – Account URI and Credentials



© Copyright KodeKloud

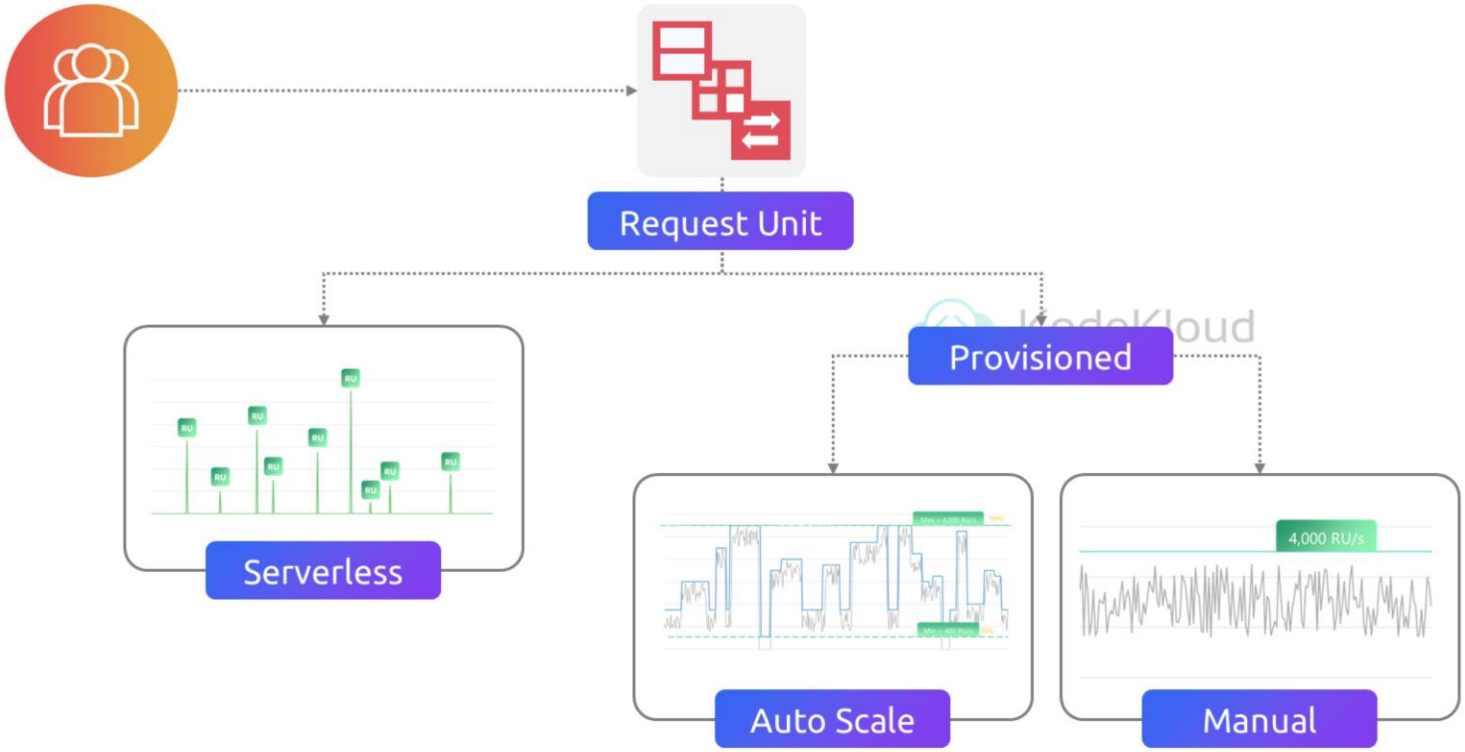
Supported APIs



- API for NoSQL: This API stores data in document format. It offers the best end-to-end experience as we have full control over the interface, service, and the SDK client libraries.
- API for MongoDB: This API stores data in a document structure, via BSON format. It is compatible with MongoDB wire protocol.
- API for Apache Cassandra: This API stores data in column-oriented schema and is wire protocol compatible with Apache Cassandra.
- API for Table: This API stores data in key/value format.

- API for Apache Gremlin: This API allows users to make graph queries and stores data as edges and vertices.
- API for PostgreSQL: Stores data either on a single node, or distributed in a multi-node configuration

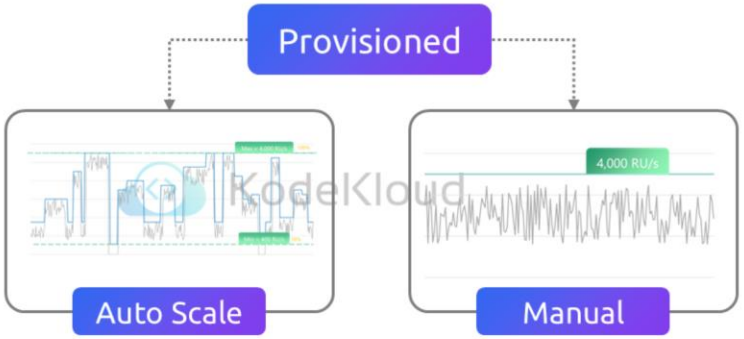
Choosing the Capacity Mode



© Copyright KodeKloud

Choosing the Capacity Mode

Best for Developers



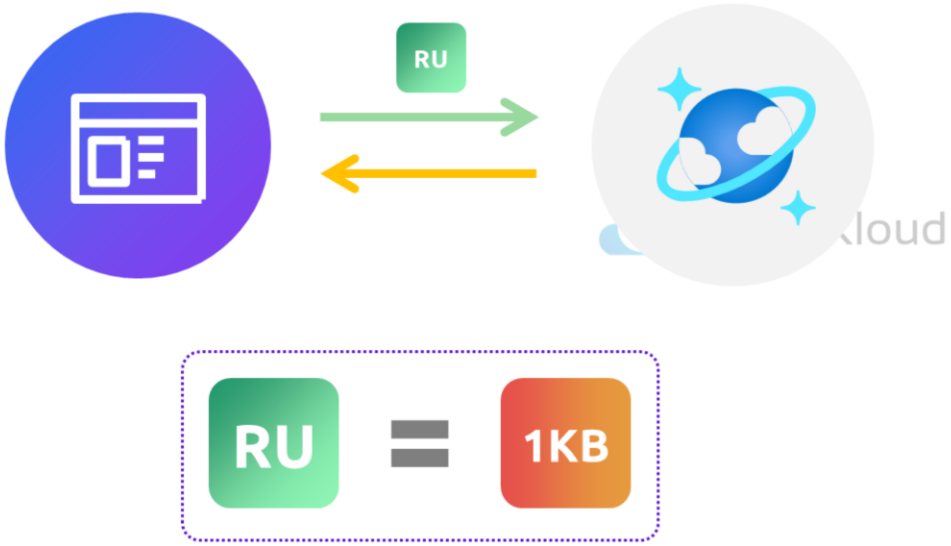
Best for Production

© Copyright KodeKloud

Azure Cosmos DB Request Units

© Copyright KodeKloud

Request Units

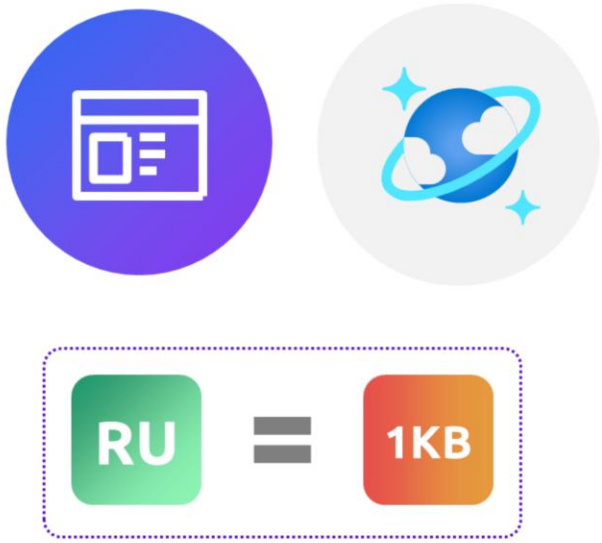


© Copyright KodeKloud

So, lets drill into how exactly the database operations you preform accrues to your dollar cost.

All operations are measured in terms of their underlying resource consumption, CPU, Memory and IO utilized and abstracted into Request Units

Request Units



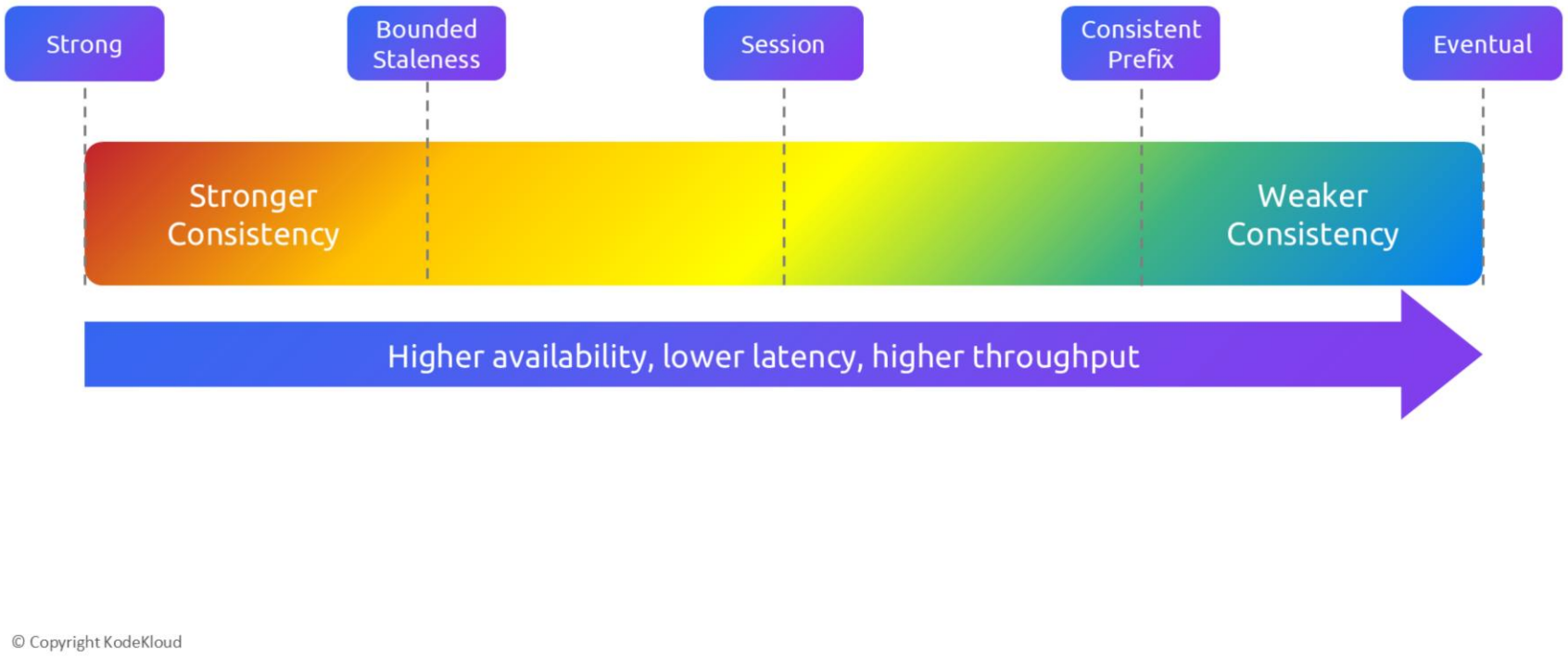
© Copyright KodeKloud

Every operation that you perform has a charge in Request Units whether it be Inserts, Reads, Queries or Deletes.
The charge for an operation is highly predictable.
If you insert, update or delete a document of a certain size it will always have the same charge.
If you run a query over the same data set it will always have the same charge.
So if you understand the operations your application needs to perform it is very easy to calculate the total required charge.

Consistency Levels

© Copyright KodeKloud

Consistency Levels



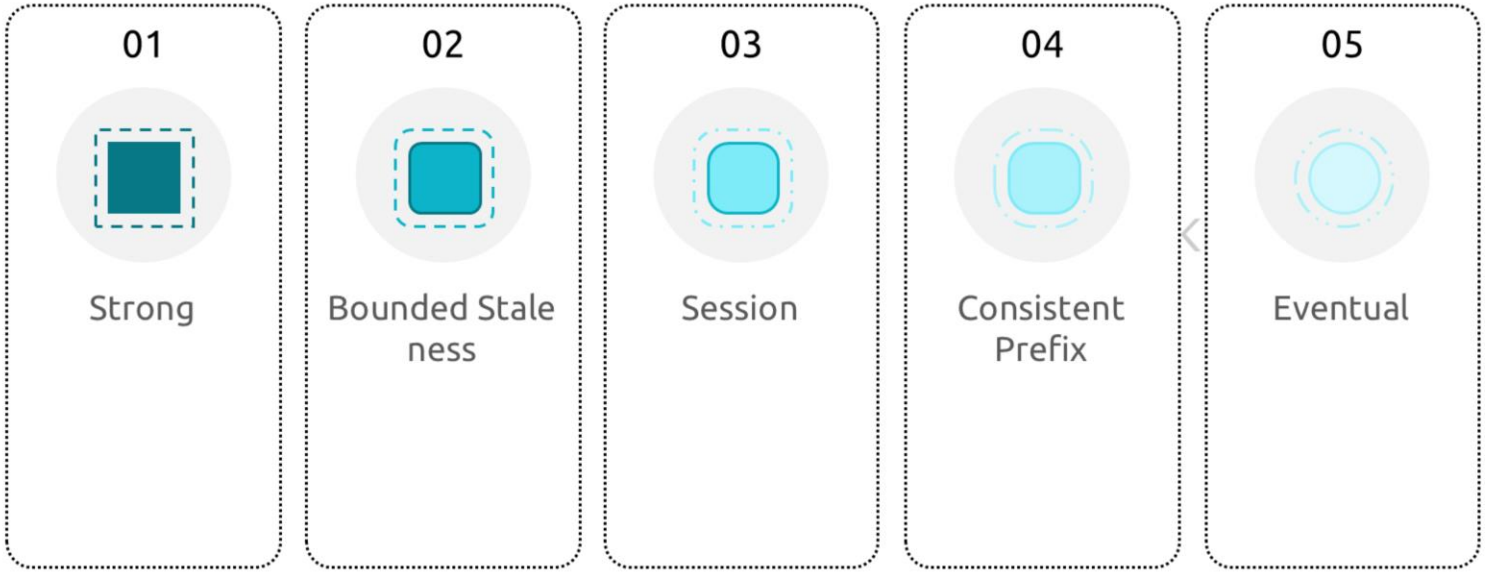
© Copyright KodeKloud

Azure Cosmos DB approaches data consistency as a spectrum of choices instead of two extremes.

Strong consistency and eventual consistency are at the ends of the spectrum, but there are many consistency choices along the spectrum.

The consistency levels are region-agnostic and are guaranteed for all operations regardless of the region from which the reads and writes are served,

Consistency Levels



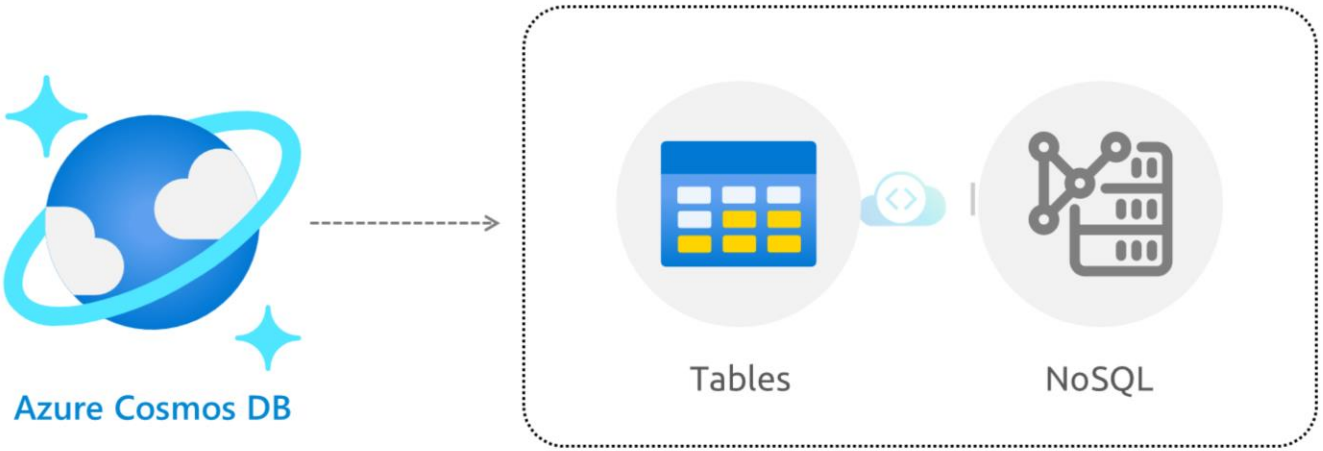
© Copyright KodeKloud

For many real-world scenarios, session consistency is optimal and it's the recommended option.

If your application requires strong consistency, it is recommended that you use bounded staleness consistency level.

If you need the highest availability and the lowest latency, then use eventual consistency level.

Choosing the Right Consistency Level



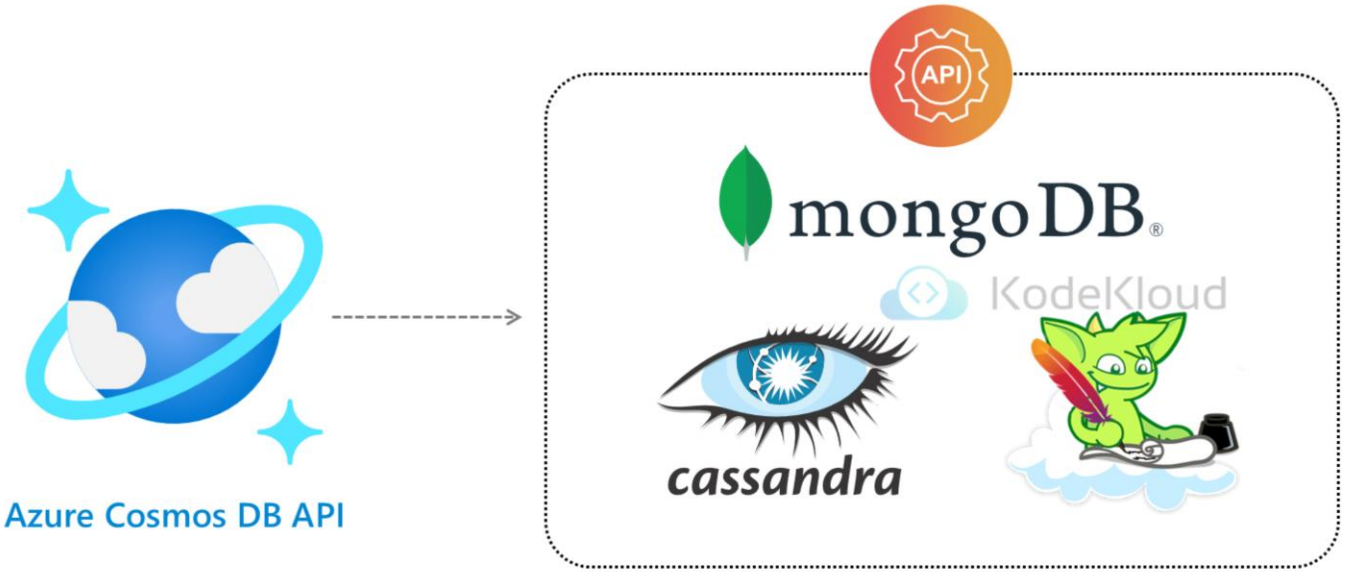
© Copyright KodeKloud

Azure Cosmos DB for NoSQL and Azure Cosmos DB for Table
For many real-world scenarios, session consistency is optimal and it's the recommended option.

If your application requires strong consistency, it is recommended that you use bounded staleness consistency level.

If you need the highest availability and the lowest latency, then use eventual consistency level.

Choosing the Right Consistency Level



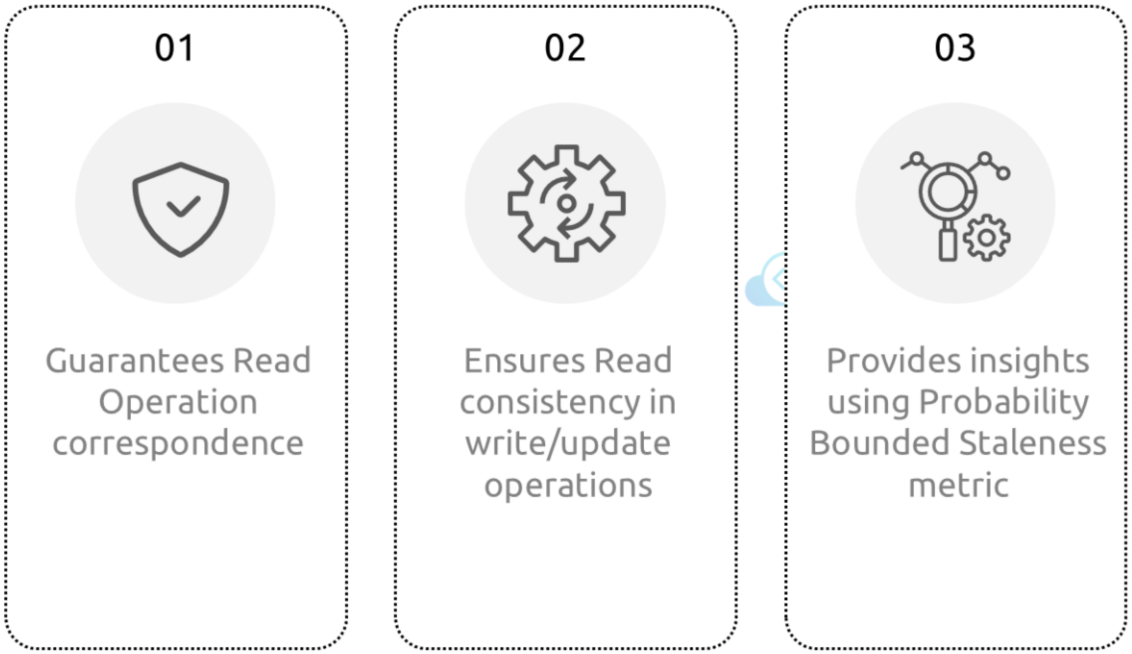
© Copyright KodeKloud

Azure Cosmos DB for Cassandra Azure Cosmos DB for MongoDB Azure Cosmos DB for Apache Gremlin
Azure Cosmos DB provides native support for wire protocol-compatible APIs for popular databases.

These include API for MongoDB, API for Apache Cassandra, and API for Apache Gremlin.

When using API for Apache Gremlin the default consistency level configured on the Azure Cosmos account is used.

Consistency Guarantees in Practice



© Copyright KodeKloud

Azure Cosmos DB for Cassandra Azure Cosmos DB for MongoDB Azure Cosmos DB for Apache Gremlin
Azure Cosmos DB provides native support for wire protocol-compatible APIs for popular databases.

These include API for MongoDB, API for Apache Cassandra, and API for Apache Gremlin.

When using API for Apache Gremlin the default consistency level configured on the Azure Cosmos account is used.

Working With Azure Cosmos DB

© Copyright KodeKloud

Introduction

01 Identify classes and methods used to create and manage Azure Cosmos DB

02 Resource management using Azure Cosmos DB .NET v3 SDK

03 Create stored procedures and user-defined functions

04 Change feed in Azure Cosmos DB

© Copyright KodeKloud

Working With Microsoft .NET SDK for Azure Cosmos DB

© Copyright KodeKloud

Creating Items

```
// Get container reference
CosmosClient client = new CosmosClient(endpoint, key);
Container container = client.GetContainer(databaseName,
collectionName);

// create anonymous type in .NET
Employee employeeRecord = new Employee {
id = "123e4567-e89b-12d3-a456-426614174000",
name = "John Doe", department = "Human Resources",
isFullTime = true, salary = $4500, yearsOfExperience = 5
};

// Upload item
Employee item = await container.CreateItemAsync(employeeRecord);
item = await container.UpsertItemAsync(employeeRecord);
```

© Copyright KodeKloud

Properties:id: Unique identifier for the employee.
name: Name of the employee.
department: The department the employee belongs to.
isFullTime: Boolean indicating full-time employment status.
salary: The employee's salary.
yearsOfExperience: Number of years the employee has been working.

Reading Items

```
// Get container reference
CosmosClient client = new CosmosClient(endpoint, key);
Container container = client.GetContainer(databaseName,
collectionName);

// Get unique fields
string id = "123e4567-e89b-12d3-a456-426614174000";
PartitionKey partitionKey = new PartitionKey("Human Resources");

// Read item using unique id
ItemResponse<Employee> response = await
container.ReadItemAsync<Employee>(id, partitionKey);

// Serialize response
Employee item = response.Resource;
```

© Copyright KodeKloud

This stored procedure takes an input parameter named documentToCreate. In this example, the parameter's value is the body of a document to be created in the current collection.

All operations are asynchronous and depend on JavaScript function callbacks. The callback function has two parameters:
One for the error object in case the operation fails.
One for the created object.

Inside the callback, users can either handle the exception or throw an error. In case a callback isn’t provided and there’s an error, the Azure Cosmos DB runtime throws an error.

In the example on the slide, the callback throws an error if the operation fails. Otherwise, it sets the id of the created document as the body of the response to the client.

Query Items

```
/
/ Get container reference
CosmosClient client = new CosmosClient(endpoint, key);
Container container = client.GetContainer(databaseName,
collectionName);

// Use SQL query language
FeedIterator<Employee> iterator =
container.GetItemQueryIterator<Employee>(
"SELECT * FROM employees e WHERE e.isFullTime = true"
);

// Iterate over results
while (iterator.HasMoreResults)
{
FeedResponse<Employee> batch = await iterator.ReadNextAsync();
foreach(Employee item in batch) { }
}
```

© Copyright KodeKloud

This stored procedure takes an input parameter named documentToCreate. In this example, the parameter's value is the body of a document to be created in the current collection.

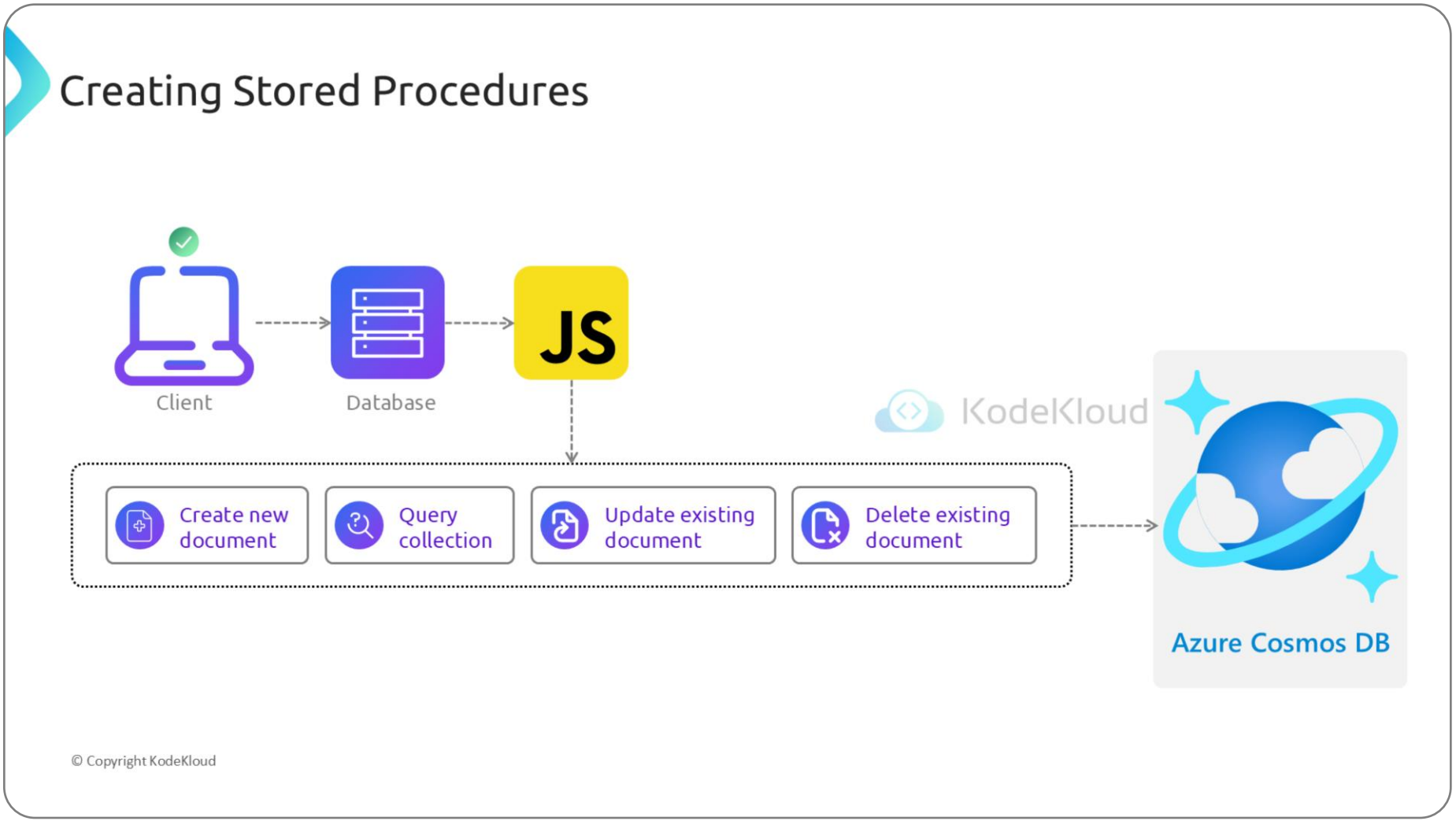
All operations are asynchronous and depend on JavaScript function callbacks. The callback function has two parameters:
One for the error object in case the operation fails.
One for the created object.

Inside the callback, users can either handle the exception or throw an error. In case a callback isn’t provided and there’s an error, the Azure Cosmos DB runtime throws an error.

In the example on the slide, the callback throws an error if the operation fails. Otherwise, it sets the id of the created document as the body of the response to the client.

Creating Stored Procedures

© Copyright KodeKloud



In a typical database, a transaction can be defined as a sequence of operations that are performed as a single, logical unit of work. Each transaction provides atomicity, consistency, isolation, and durability (ACID) guarantees. Let's understand ACID in some detail.

ACID:
Atomicity guarantees that all the work done inside a transaction is treated as a single unit where either all of it is committed or none is.

Consistency makes sure that the data is always in a good internal state across transactions.
Isolation guarantees that no two transactions interfere with each other. Generally, most commercial systems provide multiple isolation levels that can be used as per the application needs.
Durability ensures that any change that is committed in the database will always be present.

Because requests made within stored procedures and triggers run in the same scope of a database session, the services guarantee ACID for all operations that are part of a single stored procedure trigger.

This diagram illustrates the process of running a stored JavaScript procedure on Azure Cosmos DB servers and common tasks that might run within the stored procedures' scoped transaction, such as creating, querying, updating, or deleting documents.

Creating Stored Procedures

```
function createSampleDocument(documentToCreate) {
    var context = getContext();
    var collection = context.getCollection();
    var accepted = collection.createDocument(
        collection.getSelfLink(),
        documentToCreate,
        function (error, documentCreated) {
            context.getResponse().setBody(documentCreated.id)
        }
    );
    if (!accepted) return;
}
```

© Copyright KodeKloud

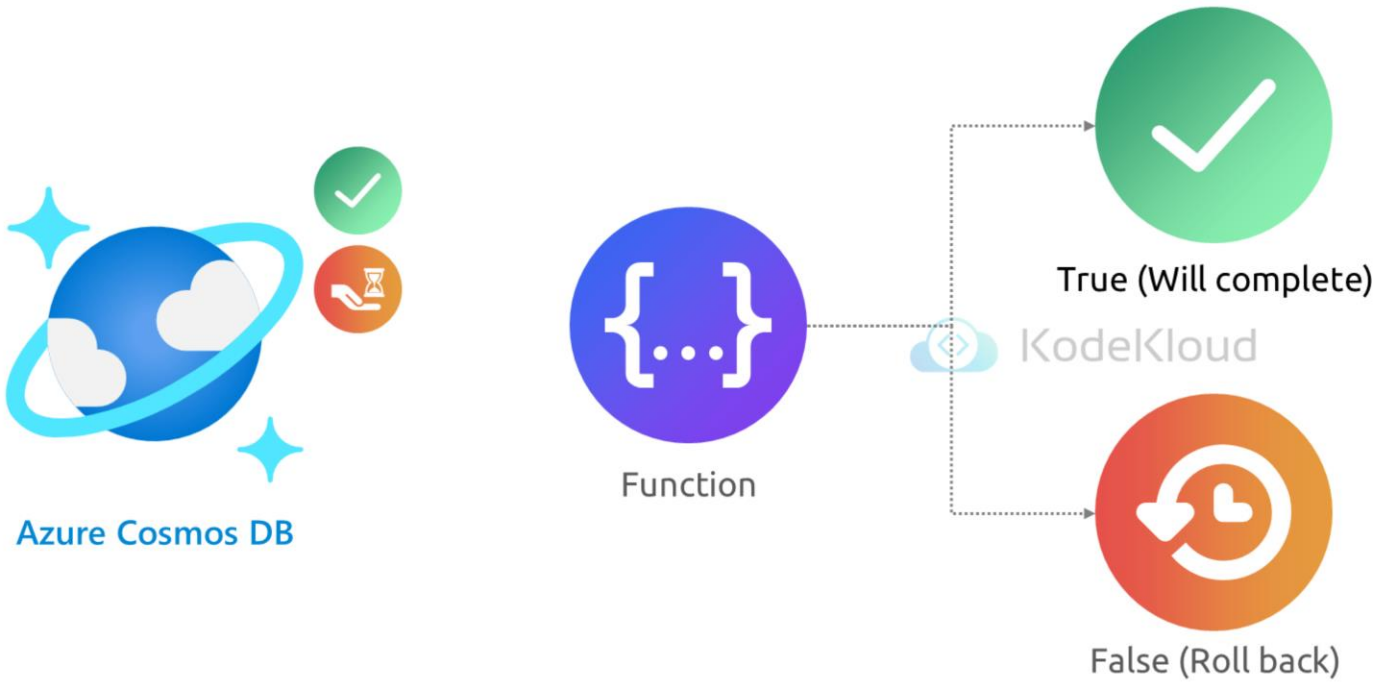
This stored procedure takes an input parameter named documentToCreate. In this example, the parameter's value is the body of a document to be created in the current collection.

All operations are asynchronous and depend on JavaScript function callbacks. The callback function has two parameters:
One for the error object in case the operation fails.
One for the created object.

Inside the callback, users can either handle the exception or throw an error. In case a callback isn’t provided and there’s an error, the Azure Cosmos DB runtime throws an error.

In the example on the slide, the callback throws an error if the operation fails. Otherwise, it sets the id of the created document as the body of the response to the client.

Creating Stored Procedures



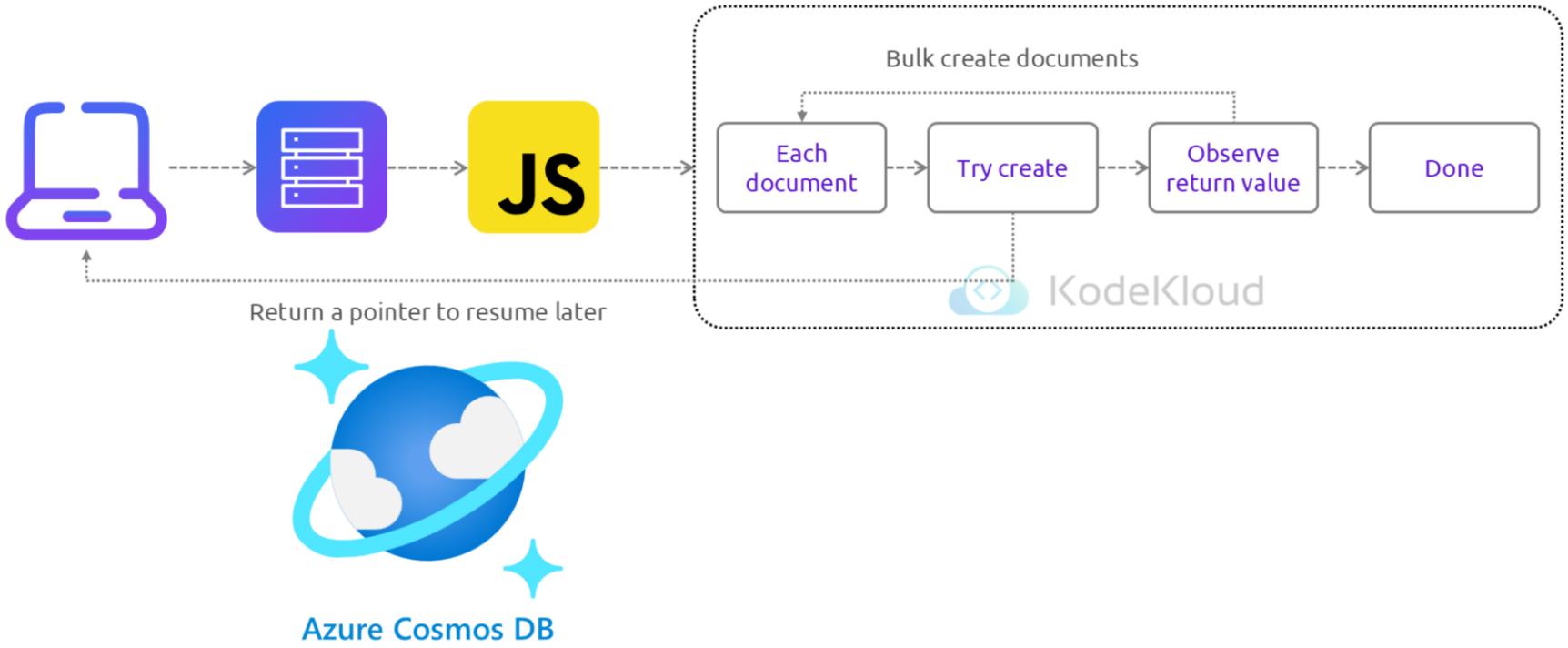
© Copyright KodeKloud

Specifically, stored procedures have a limited amount of time to run on the server

If an operation doesn't complete within this time limit, the transaction is rolled back.

All functions under the collection object (for create, read, replace, and delete documents and attachments) return a Boolean value that represents whether that operation will complete or not.

Creating Stored Procedures

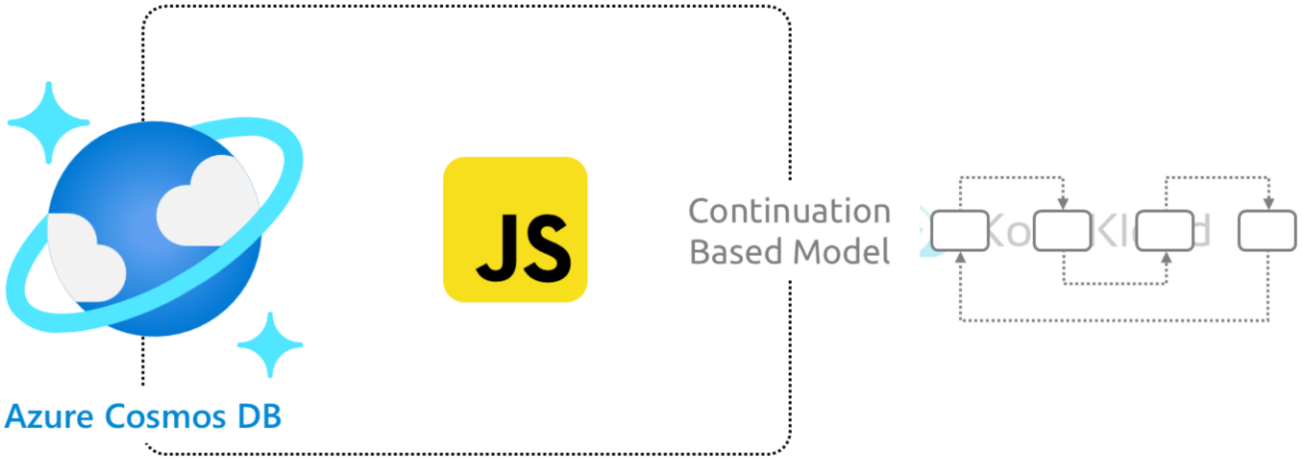


© Copyright KodeKloud

Functions can return any value. You can use this value to resume long-running functions after the server-specified time is over. For example, you might have a function to upload millions of documents. If the server-specified time is over before all documents upload, you can return a pointer to indicate how far your function progressed in the document upload. This will make it easier to resume processing in a new function iteration.

Managing Large Data with Cosmos DB Continuation Token

Ensuring Efficiency and Integrity in Bulk Operations

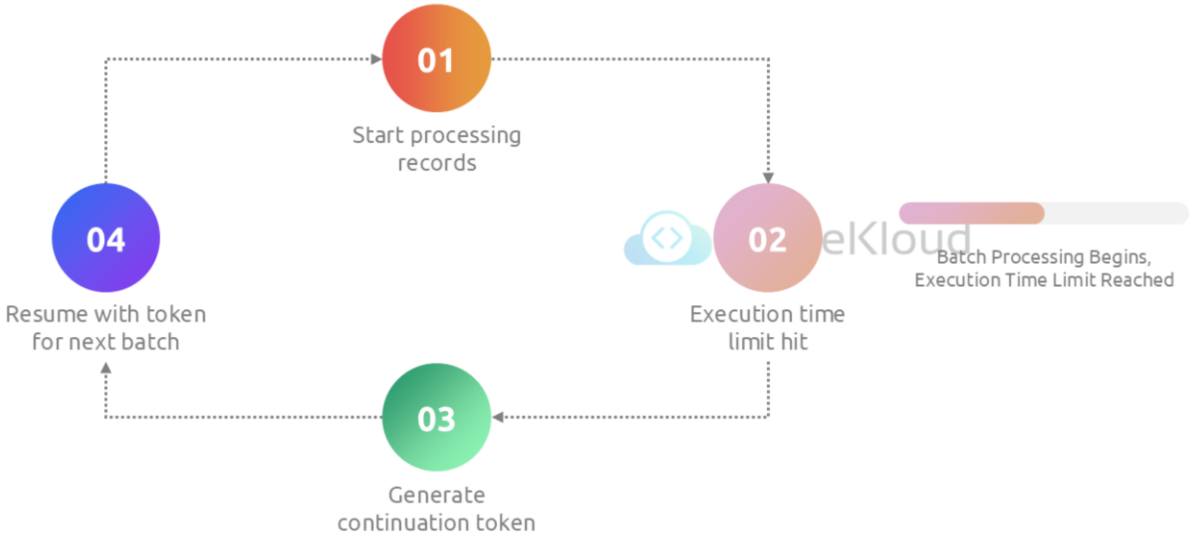


© Copyright KodeKloud

Cosmos DB allows for transaction continuation, which is especially useful when dealing with operations that might exceed the bounded execution time JavaScript functions When running in Cosmos, DING can implement a continuation based model that allows the transaction to be broken into smaller chunks The store procedure can return something called a continuation token It's, it's more like a pointer that helps pick up where it left off in a future transaction This way the process can batch documents for bulk creation or update while ensuring transactional integrity

Managing Large Data with Cosmos DB Continuation Token

Ensuring Efficiency and Integrity in Bulk Operations



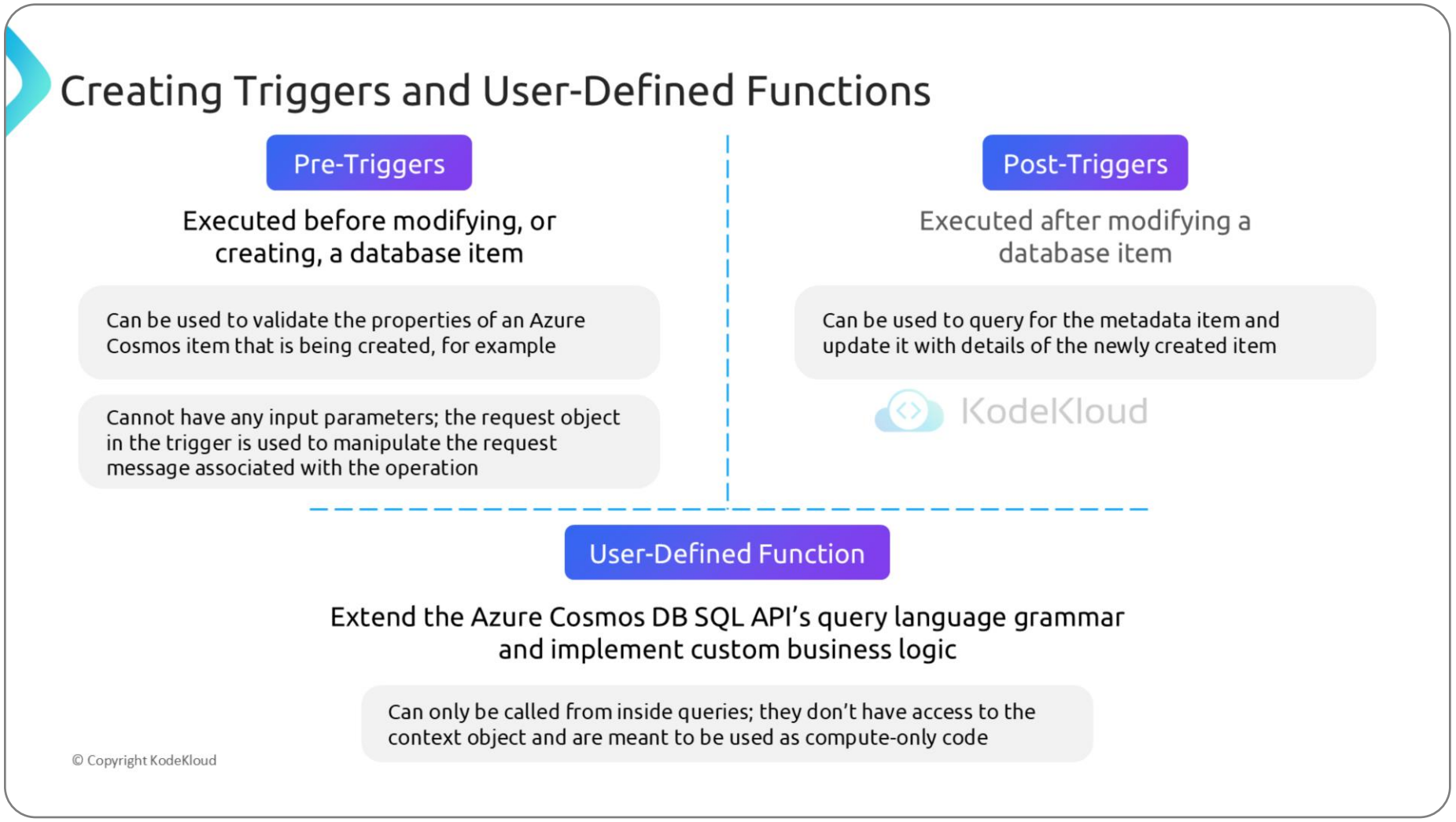
© Copyright KodeKloud

For example, let's say we want to update multiple records in our airport dataset Each document is processed in batches A token is generated for where the system leaves off in the next batch When the operation receives from the token, it'll update the next set of airport records Let's imagine you're updating the airport name for a batch of airports You would process the first few airports, the data sets, you hit the execution time limit Once you hit that limit, what you'll do is you'll return a continuation token After the first batch is done, then you can use this token to continue updating the remaining airports in a new session So this ensures efficiency and avoid timeouts when you're working with larger data sets Now, by employing these techniques, you can manage bulk operations efficiently and ensure that operations, especially on larger

data sets, don't exceed time limits while maintaining data integrity So now let's go to Azure Portal Try to write a small store procedure and see how we can work with that store procedure

Creating Triggers and User-Defined Functions

© Copyright KodeKloud



Pre-triggers: Executed before modifying, or creating, a database item.

Can be used to validate the properties of an Azure Cosmos item that is being created, for example.

Pre-triggers cannot have any input parameters. The request object in the trigger is used to manipulate the request message associated with the operation.

Post-triggers: Executed after modifying a database item

Can be used to query for the metadata item and updates it with details about the newly created item.

User-defined function: User-defined functions extend the Azure Cosmos DB SQL API's query language grammar and

implement custom business logic.

Can only be called from inside queries. They don’t have access to the context object and are meant to be used as compute-only code.

Creating Triggers and User-Defined Functions

Change feed in Azure Cosmos DB is a persistent record of changes to a container in the order they occur.

Work with Azure Cosmos DB change feed using either a **push model** or a pull model.

Azure Functions Azure Cosmos DB triggers

The change feed processor library

© Copyright KodeKloud

Change feed in Azure Cosmos DB is a persistent record of changes to a container in the order they occur.
Work with Azure Cosmos DB change feed using either a push model or a pull model.
Recommended to use the push model – no need to worry about polling the change feed for future changes.
Two ways to read from the change feed with a push model:

- Azure Functions Azure Cosmos DB triggers
- The change feed processor library

Creating Triggers and User-Defined Functions

Pre-trigger example code

```
function validateToDoItemTimestamp() {
    var context = getContext();
    var request = context.getRequest();
    // item to be created in the current operation
    var itemToCreate = request.getBody();
    // validate properties
    if (!("timestamp" in itemToCreate)) {
        var ts = new Date();
        itemToCreate["timestamp"] = ts.getTime();
    }
    // update the item that will be created
    request.setBody(itemToCreate);
}
```

KodeKloud

© Copyright KodeKloud

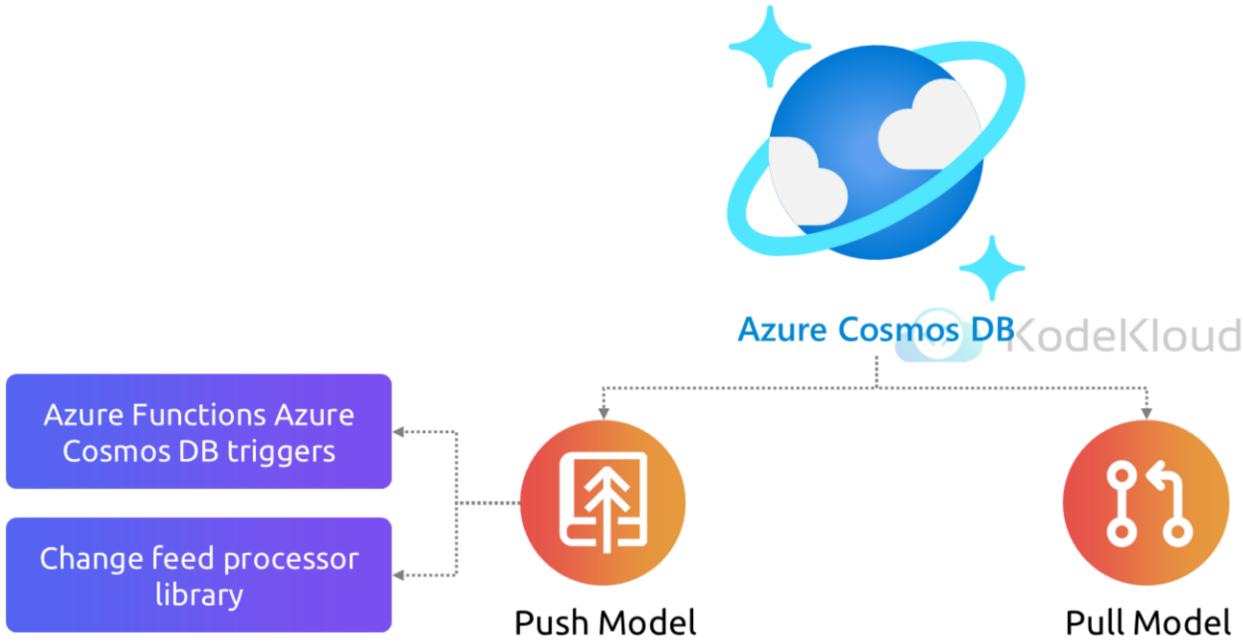
The example shows how a pre-trigger is used to validate the properties of an Azure Cosmos item that is being created, it adds a timestamp property to a newly added item if it doesn't contain one.

In the example on the slide, the callback throws an error if the operation fails. Otherwise, it sets the id of the created document as the body of the response to the client.

Change Feed in Azure Cosmos DB

© Copyright KodeKloud

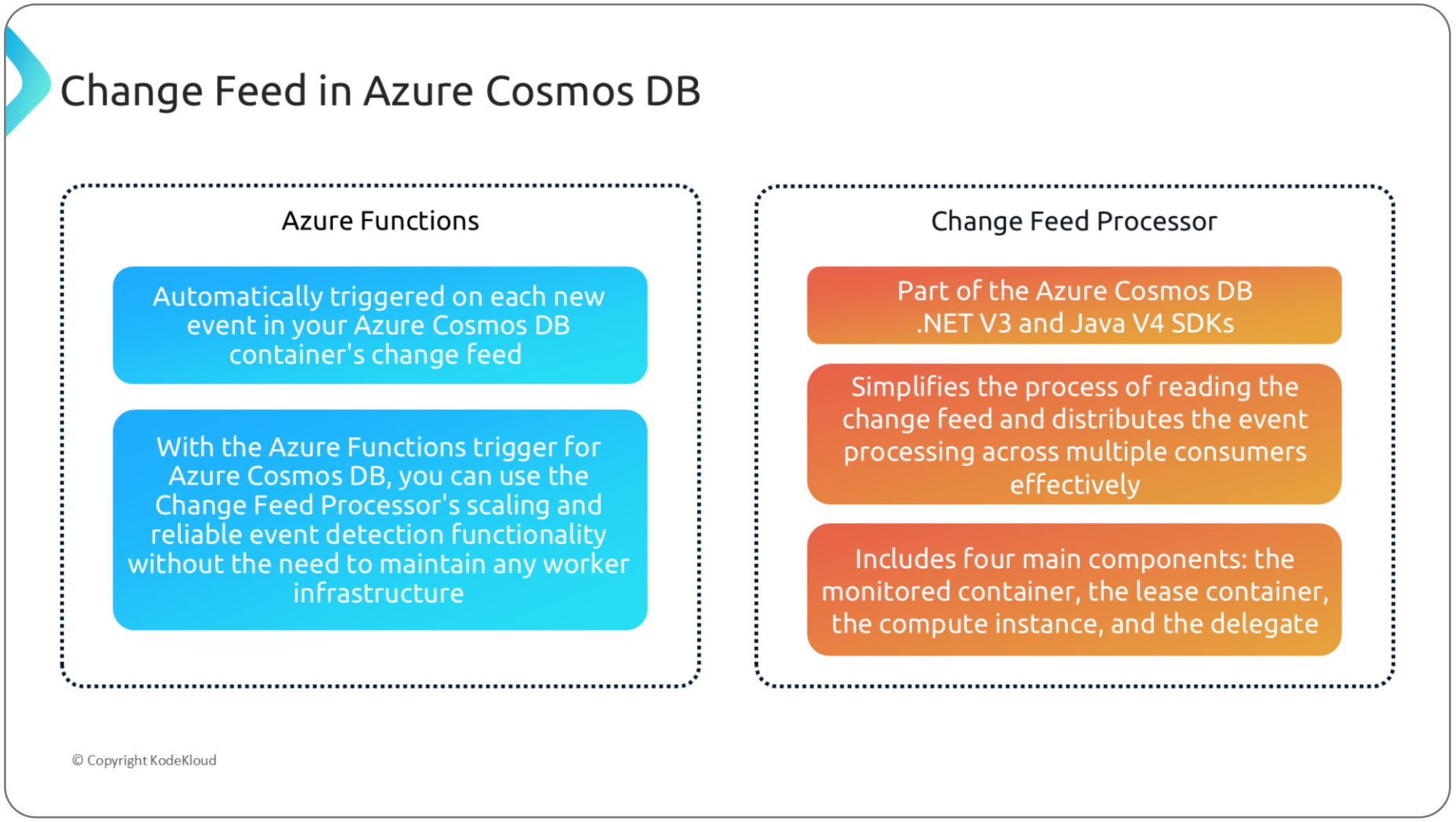
Change Feed in Azure Cosmos DB



© Copyright KodeKloud

Change feed in Azure Cosmos DB is a persistent record of changes to a container in the order they occur.
Work with Azure Cosmos DB change feed using either a push model or a pull model.
Recommended to use the push model – no need to worry about polling the change feed for future changes.
Two ways to read from the change feed with a push model:

- Azure Functions Azure Cosmos DB triggers
- The change feed processor library



Azure Functions

Automatically triggered on each new event in your Azure Cosmos DB container's change feed.
With the Azure Functions trigger for Azure Cosmos DB, you can use the Change Feed Processor's scaling and reliable event detection functionality without the need to maintain any worker infrastructure.

Change feed processor

Part of the Azure Cosmos DB .NET V3 and Java V4 SDKs.
Simplifies the process of reading the change feed and distributes the event processing across multiple consumers

effectively.
Has four main components: the monitored container, the lease container, the compute instance, and the delegate.

Change Feed in Azure Cosmos DB

Change feed processor example code

```
private static async Task<ChangeFeedProcessor> StartChangeFeedProcessorAsync(
CosmosClient cosmosClient,
IConfiguration configuration)
{
      string databaseName = configuration["SourceDatabaseName"];
      string sourceContainerName = configuration["SourceContainerName"];
      string leaseContainerName = configuration["LeasesContainerName"];
      Container leaseContainer = cosmosClient.GetContainer(databaseName, leaseContainerName);
      ChangeFeedProcessor changeFeedProcessor = cosmosClient.GetContainer(databaseName, sourceContainerName)
      .GetChangeFeedProcessorBuilder<ToDoItem>(processorName: "changeFeedSample", onChangesDelegate: HandleChangesAsync)
      .WithInstanceName("consoleHost")
      .WithLeaseContainer(leaseContainer)
      .Build();
      Console.WriteLine("Starting Change Feed Processor...");
      await changeFeedProcessor.StartAsync();
      Console.WriteLine("Change Feed Processor started.");
      return changeFeedProcessor;
}
```

© Copyright KodeKloud

Start the Change Feed Processor to listen for changes and process them with the HandleChangesAsync implementation.

KodeKloud

© Copyright KodeKloud

Visit www.kodekloud.com to learn more.