Microsoft

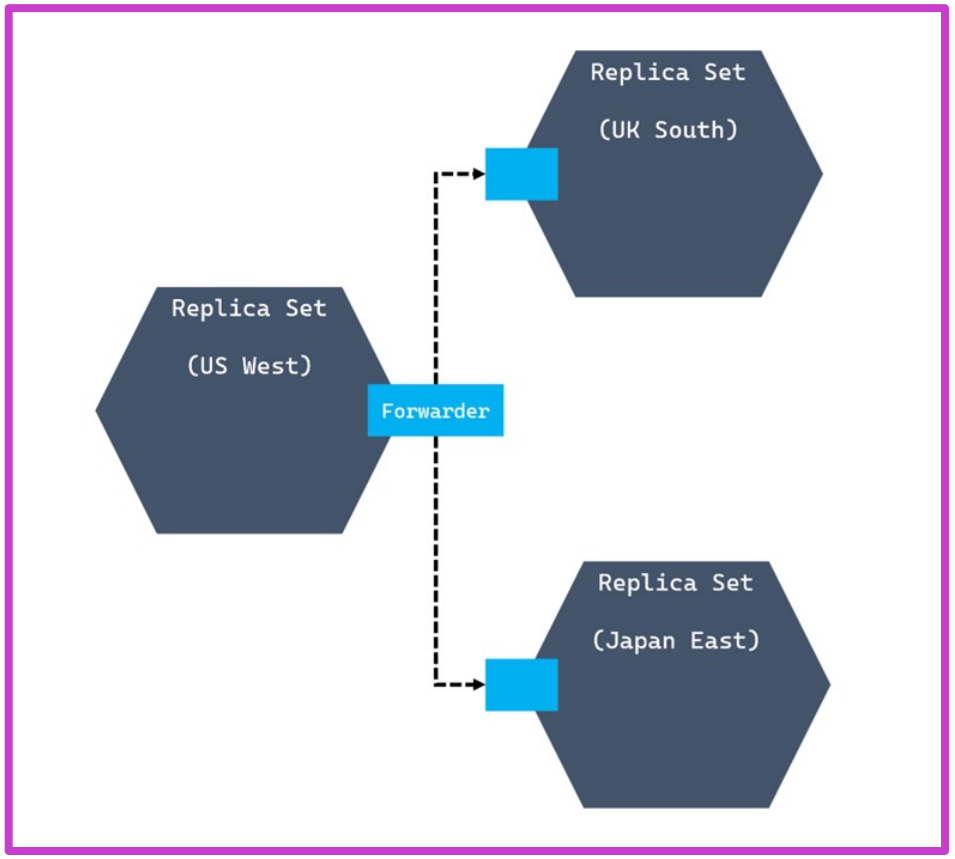
Design and implement a replication strategy for Azure Cosmos DB for NoSQL

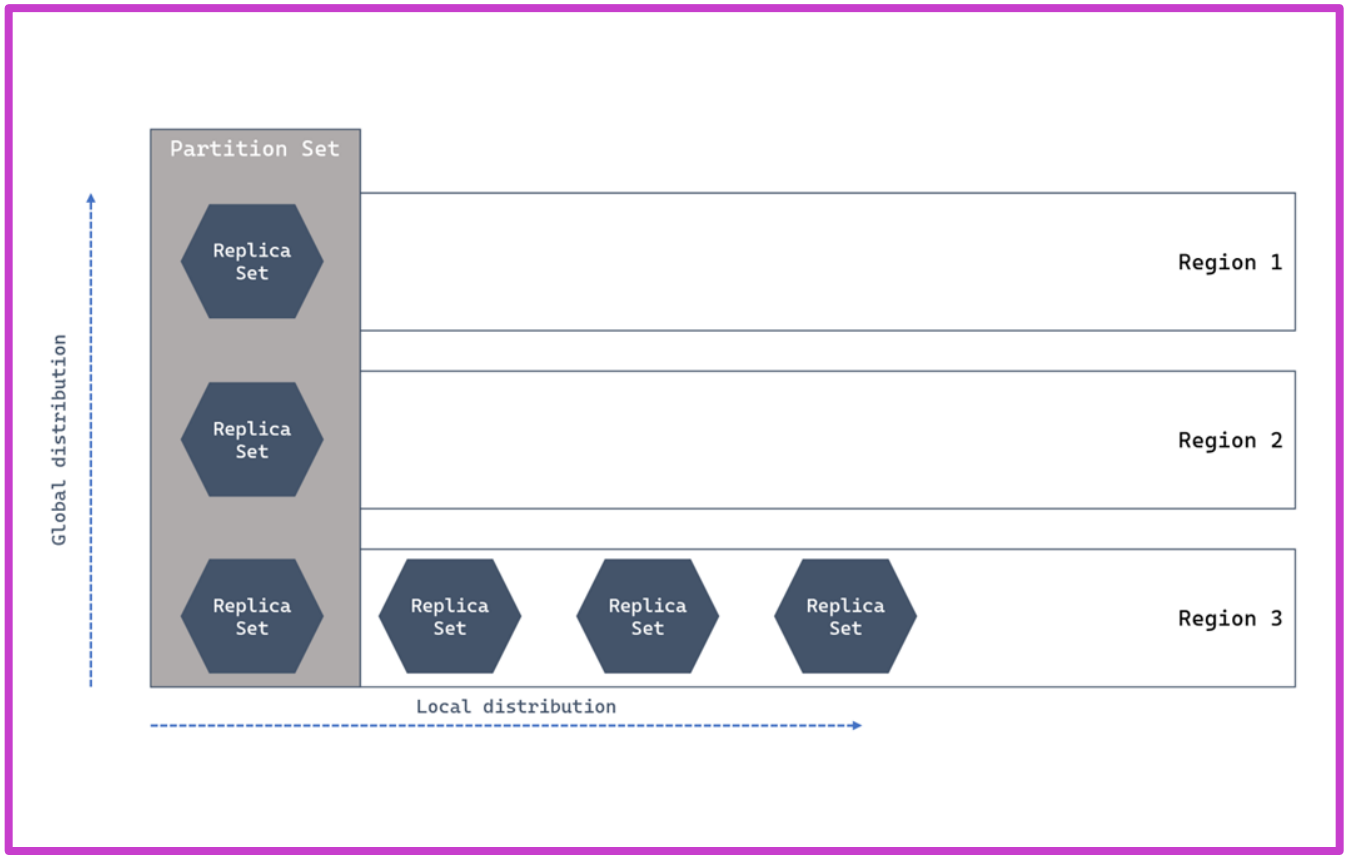
Agenda

- Configure replication and manage failovers in Azure Cosmos DB
- Use consistency models in Azure Cosmos DB for NoSQL
- Configure multi-region write in Azure Cosmos DB for NoSQL

Configure replication and manage failovers in Azure Cosmos DB

Understand replication

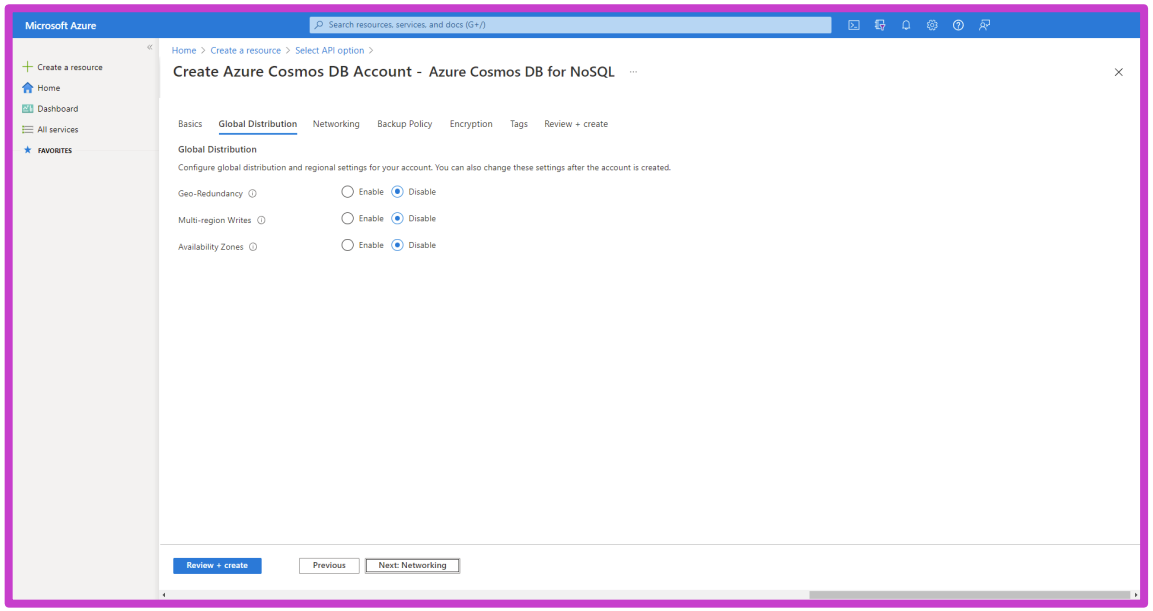




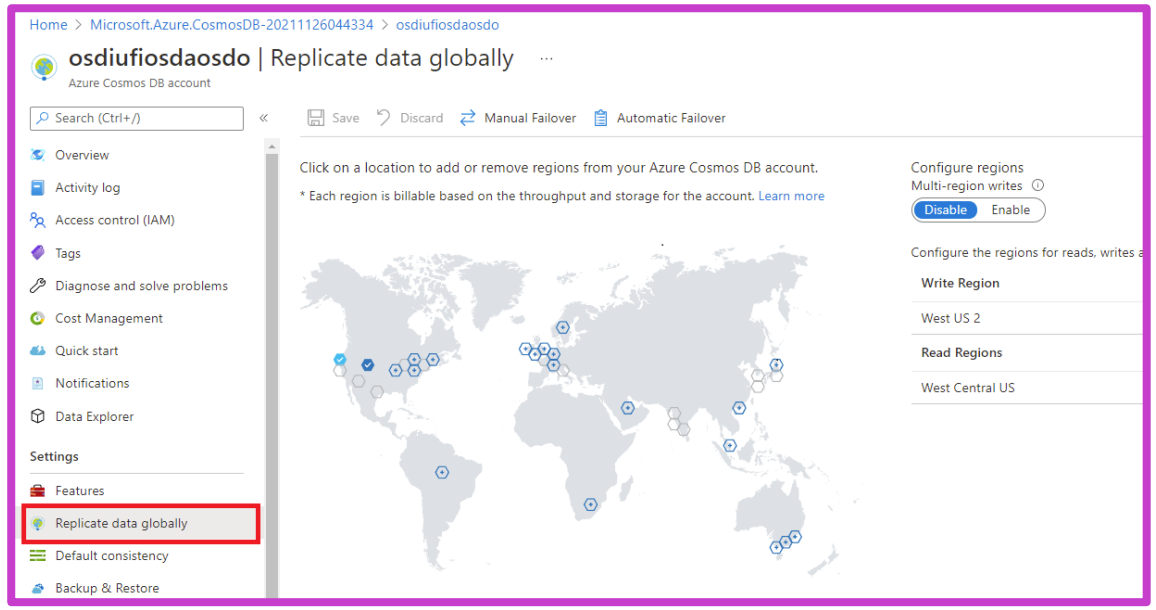
Distribute data across regions

Configuring global distribution in Azure Cosmos DB is a turnkey operation that is performed when an account is created or afterward.

Configuring geo-redundancy for a new account



Configuring geo-redundancy for an existing account

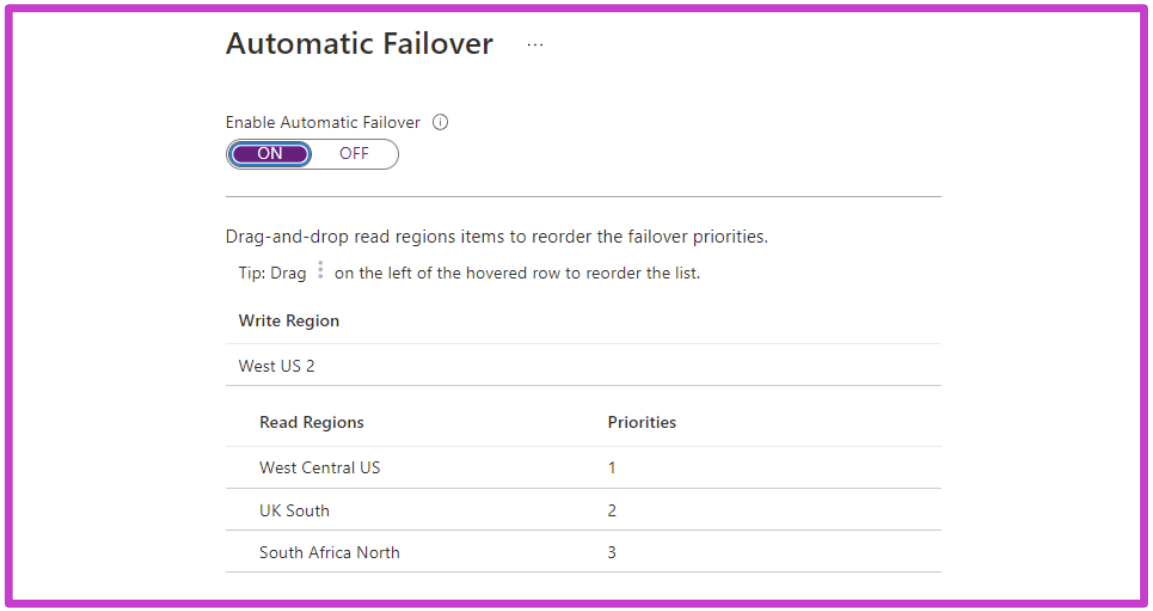


The cost of distributing data globally is **RU/s** x **# of regions**.

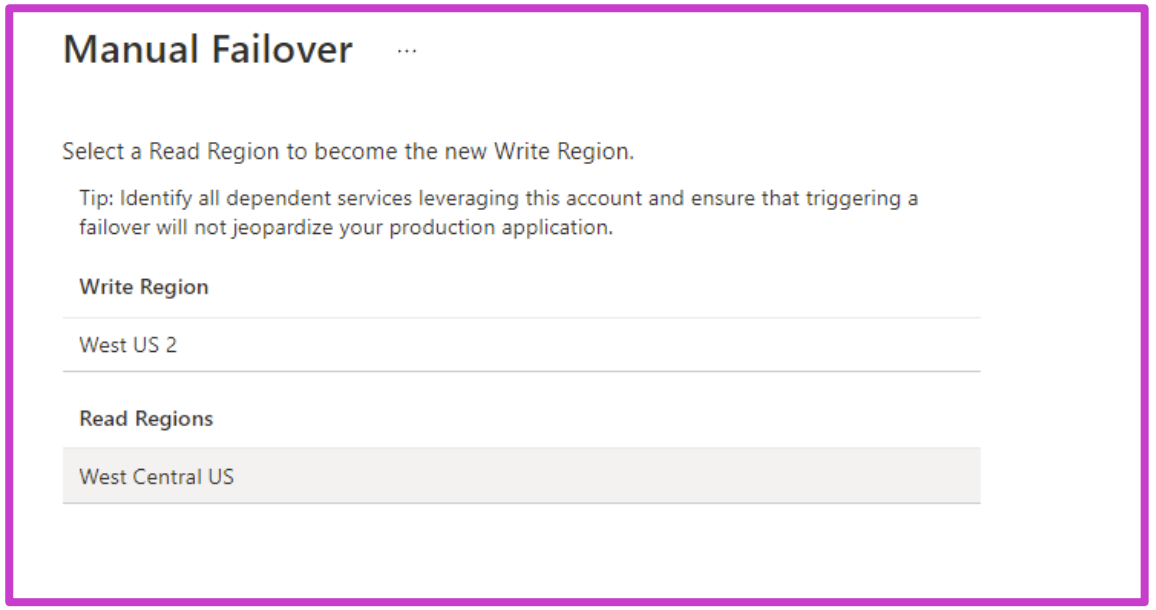
Azure Cosmos DB failovers

An automatic failover plan can transfer the write region to one of the read regions in the case of an outage.

Define automatic failover policies



Perform manual failovers



Configure SDK region

Use the *ApplicationRegion* or *ApplicationPreferredRegions* properties to configure preferred regions.

Setting an application region

```
CosmosClientOptions options = new () { ApplicationRegion = Regions.UKSouth };
CosmosClient client = new (connectionString, options);

//  Or using the CosmosClientBuilder class
CosmosClient client = new CosmosClientBuilder(connectionString)
    .WithApplicationRegion(Regions.UKSouth)
    .Build();
```

Setting a list of preferred application regions

```
List<string> regions = new() { "East Asia", "South Africa North", "West US" };
CosmosClientOptions options = new () { ApplicationPreferredRegions = regions };
CosmosClient client = new (connectionString, options);

//  Or using the CosmosClientBuilder class
CosmosClient client = new CosmosClientBuilder(connectionString)
  .WithApplicationPreferredRegions( new List<string> { Regions.EastAsia, Regions.SouthAfricaNorth,
Regions.WestUS } ) .Build();
```

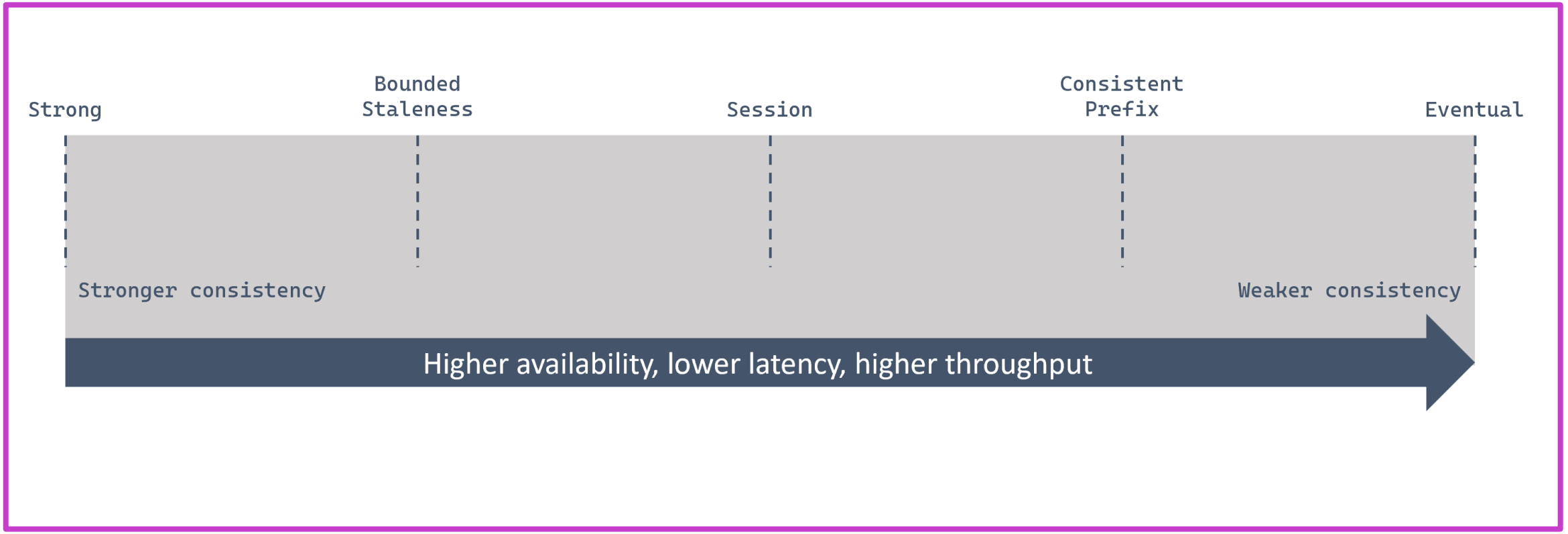
Lab – Connect to different regions with the Azure Cosmos DB for NoSQL SDK

- Prepare your development environment
- Create an Azure Cosmos DB for NoSQL account
- Connect to the Azure Cosmos DB for NoSQL account from the SDK
- Configure the .NET SDK with a preferred region list

Use consistency models in Azure Cosmos DB for NoSQL

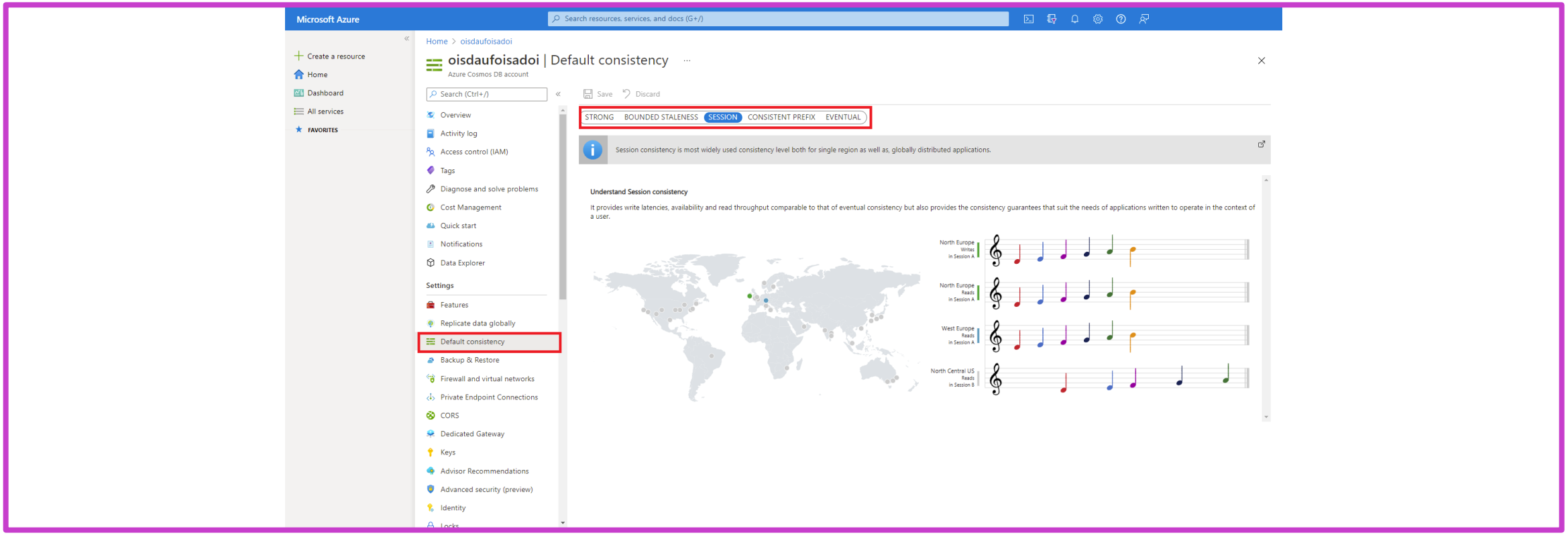
Understand consistency models

In a distributed database system, tradeoffs are often made between highly consistent data with extended latency and speedy data operations that may not be consistent immediately.



Configure default consistency model in the portal

In the Azure portal, the Default consistency pane is used to configure a new default consistency level for the entire account.



Change consistency model with the SDK

Using the *ItemRequestOptions* class, you can relax the current default consistency level to a weaker one.

```
// Set the item request Consistency Level option

ItemRequestOptions options = new()
{
    ConsistencyLevel = ConsistencyLevel.Eventual
};

// Assign the option to the create item operation
var item = new Product
{
    id = $"{Guid.NewGuid()}",
    categoryId = $"{Guid.NewGuid()}",
    name = "Reflective Handlebar" ‘
};

await container.CreateItemAsync<Product>(item, requestOptions: options);
```

Use session tokens

Using the .NET SDK classes, the session token can be manually extracted and passed back to the Azure Cosmos DB resource.

```
// Create an item with an Item Response
ItemResponse<Product> response = await container.CreateItemAsync<Product>(item);

// Get the session token from the item response
string token = response.Headers.Session;

// Set the item request option session token to the previous token
ItemRequestOptions options = new()
{
    SessionToken = token
};

// Use the token on the new request
ItemResponse<Product> readResponse = container.ReadItemAsync<Product>(id, partitionKey,
requestOptions: options);)
```

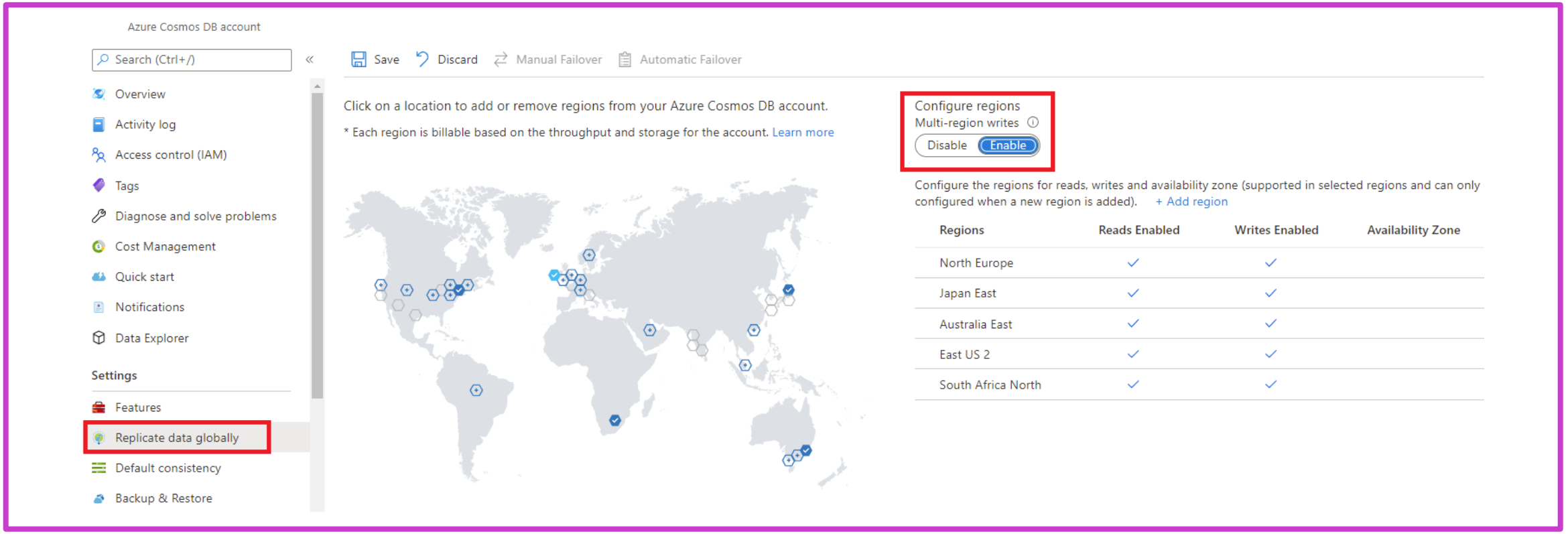
Lab – Configure consistency models in the portal and the Azure Cosmos DB for NoSQL SDK

- Prepare your development environment
- Create an Azure Cosmos DB for NoSQL account
- Connect to the Azure Cosmos DB for NoSQL account from the SDK
- Configure consistency level for a point operation

Configure multi-region write in Azure Cosmos DB for NoSQL

Understand multi-region write

With Azure Cosmos DB, every region supports both writes and reads.



Understand conflict resolution policies

Azure Cosmos DB's multi-region write feature has automatic conflict management built in. The default policy is known as Last Write Wins that uses the _ts property by default.

Replace the default _*ts*_ property configuring *any numeric property* **as a *conflict resolution path* on the .NET SDK**

```
Database database = client.GetDatabase("cosmicworks");

// Define a default custom conflict resolution path as /metadata/sortableTimestamp.
ContainerProperties properties = new("products", "/categoryId")
{
    ConflictResolutionPolicy = new ConflictResolutionPolicy()
    {
        Mode = ConflictResolutionMode.LastWriterWins,
        ResolutionPath = "/metadata/sortableTimestamp",
    }
};

// Note: You can only set a conflict resolution policy on newly created containers.
Container container = database.CreateContainerIfNotExistsAsync(properties);
```

Create a custom conflict resolution policy step 1: JavaScript SP

Create a custom conflict resolution policy when you wish to write your own logic to resolve conflicts between items. Let's first define the custom JavaScript SP that resolves the conflicts.

```
function <function-name>(incomingItem, existingItem, isTombstone, conflictingItems)
```

Custom conflict resolution JavaScript stored procedure example that uses the *`/metadata/sortableTimestamp`* path.

```
function resolveConflicts(incomingItem, existingItem, isTombstone, conflictingItems)
{
    if (!incomingItem) { if (existingItem) { __.deleteDocument(existingItem._self, {}); } }
    else if (isTombstone) { }
    else { if (existingItem) { if (incomingItem.metadata.sortableTimestamp > existingItem.metadata.sortableTimestamp) { return; } }
           var i;
           for (i = 0; i < conflictingItems.length; i++)
           { if (incomingItem.metadata.sortableTimestamp > conflictingItems[i].metadata.sortableTimestamp) { return; } }
           delete (conflictingItems, incomingItem, existingItem);
         }
     function delete (documents, incoming, existing)
     {
         if (documents.length > 0)
         { __.deleteDocument(documents[0]._self, {}, function (err, responseOptions) { documents.shift();
           delete (documents, incoming, existing); });}
         else if (existing)
         { __.replaceDocument(existing._self, incoming); }
         else { __.createDocument(collection.getSelfLink(), incoming); }
      }
}
```

Create a custom conflict resolution policy step 2: .NET SDK

Once the JavaScript SP code has been defined, let's assume it was written to the *resolver.js* file.

```
Database database = client.GetDatabase("cosmicworks");

// Define the custom conflict resolution path as /metadata/sortableTimestamp.
ContainerProperties properties = new("products", "/categoryId")
{
    ConflictResolutionPolicy = new ConflictResolutionPolicy()
    {
        Mode = ConflictResolutionMode.Custom,
        ResolutionProcedure = string.Format("dbs/{0}/colls/{1}/sprocs/{2}",
                                            this.databaseName,
                                            this.udpCollectionName,
                                            "resolver")

    }
};

// Note: You can only set a conflict resolution policy on newly created containers.
Container container = database.CreateContainerIfNotExistsAsync(properties);

StoredProcedure ResolverSP = container.Scripts.CreateStoredProcedureAsync(
          new StoredProcedureProperties("resolver", File.ReadAllText(@"resolver.js"))
);
```

Lab – Connect to a multi-region write account with the Azure Cosmos DB for NoSQL SDK

- Prepare your development environment
- Create an Azure Cosmos DB for NoSQL account
- Connect to the Azure Cosmos DB for NoSQL account from the SDK
- Configure write region for the SDK