TypeText must be input into **Notepad** and then copied into **VS Code** in order to maintain formatting.

Do not click nested Typetext icons. Doing so will produce the wrong output.

```
public class Program
{
    private const string blobServiceEndpoint = "";
    private const string storageAccountName = "";
    private const string storageAccountKey = "";

    public static async Task Main(string[] args)
    {
    }

}
```

&lt in a Typetext Box should be changed to <

&gt in a Typetext Box should be changed to >

At the end of this lab, you can skip the **Clean Up** exercise directing you to remove the resources from your Subscription or Resource Group(s). The clean up is handled automatically, after ending your lab.

more...

Lab: Constructing a polyglot data solution

Student lab manual

Lab scenario

You have been assigned the task of updating your company's existing retail web application to use more than one data service in Microsoft Azure. Your company's goal is to take advantage of the best data service for each application component. After conducting thorough research, you decide to migrate your inventory database from Azure SQL Database to Azure Cosmos DB.

Objectives

After you complete this lab, you will be able to:

- Create instances of various database services by using the Azure portal.
- Write C# code to connect to SQL Database.
- Write C# code to connect to Azure Cosmos DB.

Lab setup

- Estimated time: **45 minutes**

Instructions

Before you start

Sign in to the lab virtual machine

Ensure that you're signed in to your Windows 10 virtual machine by using the following credentials:

- Username: **Admin**
- Password: **Pa55w.rd**

Review the installed applications

Find the taskbar on your Windows 10 desktop. The taskbar contains the icons for the applications that you'll use in this lab:

- Microsoft Edge
- File Explorer
- Visual Studio Code

Exercise 1: Creating database resources in Azure

Task 1: Open the Azure portal

1. Sign in to the Azure portal (https://portal.azure.com).
2. If this is your first time signing in to the Azure portal, you'll notice a dialog box offering a tour of the portal. Select the **Get Started** button to skip the tour.

Task 2: Create an Azure SQL Database server resource

1. Create a new **Azure SQL Database server (logical server)** resource with the following details:
    - Server name: **polysqlsrvr*[yourname]***
    - New resource group: **PolyglotData**
    - Server admin login: **testuser**
    - Password: **TestPa55w.rd**
    - Location: **(US) East US**
    - Allow Azure services to access server: **Yes**

    Note: At this point in the lab, we are only creating the Azure SQL logical server. We will create the Azure SQL database instance later in the lab.

    Note: Wait for Azure to finish creating the SQL server instance before you move forward with the lab. You'll receive a notification when the SQL server is created.

Task 3: Create an Azure Cosmos DB account resource

1. Create a new **Azure Cosmos DB** instance with the following details:
    - Account name: **polycosmos*[yourname]***
    - Existing resource group: **PolyglotData**
    - API: **Core (SQL)**
    - Notebooks (Preview): **Off**
    - Apply Free Tier Discount: **Do Not Apply**
    - Location: **(US) East US**
    - Account Type **Non-Production**
    - Multi-region writes: **Disable**

    Note: Wait for Azure to finish creating the Azure Cosmos DB account before you move forward with the lab. You'll receive a notification when the Azure Cosmos DB account is created.

2. Go to the blade for your newly created Azure Cosmos DB account resource, and then open the Keys pane.
3. In the Keys pane, record the value of the **PRIMARY CONNECTION STRING** text box.

    Note: You'll use these values later in this lab.

Task 4: Create an Azure Storage account resource

1. Create a new Azure Storage account with the following details:
    - Storage account name: **polystor*[yourname]***
    - Existing resource group: **PolyglotData**
    - Account kind: **StorageV2 (general purpose v2)**
    - Location: **(US) East US**
    - Replication: **Locally-redundant storage (LRS)**
    - Performance: **Standard**
    - Access tier (default): **Hot**

    Note: Wait for Azure to finish creating the storage account before you move forward with the lab. You'll receive a notification when the storage account is created.

Review

In this exercise, you created all the Azure resources that you'll need for a polyglot data solution.

Exercise 2: Import and validate data

Task 1: Upload image blobs

1. Go to the blade for the **polystor*[yourname]*** Azure Storage account that you created earlier in this lab.
2. Open the Containers pane, and then create a new container with the following settings:
    - Name: **images**
    - Public access level: **Blob (anonymous read access for blobs only)**
3. Go to the new **images** container, and then open the Properties pane.

4. In the Properties pane, record the value in the **URL** text box.

   Note: You'll use this value later in this lab.

5. Return to the blade for the **images** container.
6. Use the **Upload** button to upload the 42 individual **.jpg** image files in the **Allfiles (F):\Allfiles\Labs\04\Starter\Images** folder on your lab machine.

   Note: We recommended that you enable the **Overwrite if files already exist** option.

Task 2: Upload an SQL .bacpac file

1. Go back to the blade for the **polystor*[yourname]*** Azure Storage account, and then open the Containers pane again.
2. Create a new container with the following settings:
   - Name: **databases**
   - Public access level: **Private (no anonymous access)**
3. Go to the new **databases** container.
4. Use the **Upload** button to upload the **AdventureWorks.bacpac** file in the **Allfiles (F):\Allfiles\Labs\04\Starter** folder on your lab machine.

   Note: We recommended that you enable the **Overwrite if files already exist** option.

Task 3: Import an SQL database

1. Go to the blade for the **polysqlsrvr*[yourname]*** SQL server resource that you created earlier in this lab.
2. Import the database from your Azure Storage account into the SQL server instance by using the following details:
   - Storage account: **polystor*[yourname]***
   - Database backup blob: **databases\AdventureWorks.bacpac**
   - Database name: **AdventureWorks**
   - Server admin login: **testuser**
   - Password: **TestPa55w.rd**

   Note: Wait for the database to be created before you continue with this lab. If you receive a firewall-related error on the import step, it means you did not correctly configure the **Allow Azure services to access server** setting on your SQL Server earlier in the lab. Review your settings, delete the empty **AdventureWorks** database, and then attempt your import again.

Task 4: Use an imported SQL database

1. Go back to the blade for the **polysqlsrvr*[yourname]*** SQL server resource.
2. Open the Firewalls and virtual networks pane.
3. Add your current client IP address to the list of allowed IP addresses, and then save the list.
4. Go to the blade for the **AdventureWorks** SQL database resource that you recently imported.
5. Open the Connection strings pane, and then record the value of the **ADO.NET (SQL Authentication)** connection string.
6. Update the connection string that you recorded by replacing the placeholder values for *your_username* and *your_password*

   Note: For example, if your connection string was originally
   ```
   Server=tcp:polysqlsrvrinstructor.database.windows.net,1433;InitialCatalog=AdventureWorks;User ID={your_username};Password={your_password};
   ```
   , your updated connection string will be
   ```
   Server=tcp:polysqlsrvrinstructor.database.windows.net,1433;InitialCatalog=AdventureWorks;User ID=testuser;Password=Te
   ```

7. Go back to the blade for the **AdventureWorks** SQL database resource.
8. Open the Query editor pane, and then sign in by using the following credentials:
   - Username: **testuser**
   - Password: **TestPa55w.rd**
9. Run the following query, and then observe the results:

   ```
   SELECT * FROM AdventureWorks.dbo.Models
   ```

   Note: This query will return a list of models from the home page of the web application.

10. Run this additional query, and then observe the results:

    ```
    SELECT * FROM AdventureWorks.dbo.Products
    ```

    Note: This query will return a list of products that are associated with each model.

Review

In this exercise, you imported all the resources that you'll use with your web application.

Exercise 3: Open and configure a .NET web application

Task 1: Open and build the web application

1. Using Visual Studio Code, open the solution folder found at **Allfiles (F):\Allfiles\Labs\04\Starter\AdventureWorks**.
2. Using a terminal, build the .NET solution:

   ```
   dotnet build
   ```

   Note: The **dotnet build** command will automatically restore any missing NuGet packages prior to building all projects in the folder.

3. Close the current terminal.

Task 2: Update the SQL connection string

1. Open the **AdventureWorks.Web/appsettings.json** file in Visual Studio Code.
2. Replace the value of the *ConnectionStrings.AdventureWorksSqlContext* property with the **ADO.NET (SQL Authentication) connection string** of the SQL database that you recorded earlier in this lab.

   Note: It's important that you use your updated connection string here. The original connection string copied from the portal won't have the username and password necessary to connect to the SQL database.

3. Save the **appsettings.json** file.

Task 3: Update the blob base URL

1. Replace the value of the *Settings.BlobContainerUrl* property with the *URL* property of the **Azure Storage** blob container named **images** that you recorded earlier in this lab.
2. Save the **appsettings.json** file.

Task 4: Validate the web application

1. Using a terminal, change your context to the **AdventureWorks.Web** folder:

   ```
   cd .\AdventureWorks.Web\
   ```

2. Using the same terminal, run the ASP.NET web application project:

   ```
   dotnet run
   ```

   Note: The **dotnet run** command will automatically build any changes to the project and then start the web application without a debugger attached. The command will output the URL of the running application and any assigned ports.

3. Open the Microsoft Edge browser.
4. In the open browser window, browse to the web application that's hosted at **localhost** on port **5000**.

   Note: The URL is http://localhost:5000.

5. In the web application, observe the list of models displayed from the front page.
6. Find the **Water Bottle** model, and then select **View Details**.
7. From the **Water Bottle** product detail page, find **Add to Cart**, and then observe that the checkout functionality is currently disabled.
8. Close the browser window that's displaying your web application.
9. Return to the **Visual Studio Code** window.
10. Close the current terminal.

Review

In this exercise, you configured your ASP.NET web application to connect to your resources in Azure.

Exercise 4: Migrating SQL data to Azure Cosmos DB

Task 1: Create a migration project

1. Using a terminal, create a new .NET console project named **AdventureWorks.Migrate** in a folder with the same name:

   ```
   dotnet new console --name AdventureWorks.Migrate
   ```

   Note: The **dotnet new** command will create a new **console** project in a folder with the same name as the project.

2. Using the same terminal, add a reference to the existing **AdventureWorks.Models** project:

```
dotnet add .\AdventureWorks.Migrate\AdventureWorks.Migrate.csproj reference .\AdventureWorks.Models\AdventureWorks.Mo
```

Note: The **dotnet add reference** command will add a reference to the model classes contained in the **AdventureWorks.Models** project.

3. Using the same terminal, add a reference to the existing **AdventureWorks.Context** project:

```
dotnet add .\AdventureWorks.Migrate\AdventureWorks.Migrate.csproj reference .\AdventureWorks.Context\AdventureWorks.C
```

Note: The **dotnet add reference** command will add a reference to the context classes contained in the **AdventureWorks.Context** project.

4. Using the same terminal, change your context to the **AdventureWorks.Migrate** folder:

```
cd .\AdventureWorks.Migrate\
```

5. Using the same terminal, import version 3.0.1 of **Microsoft.EntityFrameworkCore.SqlServer**from NuGet:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 3.0.1
```

Note: The **dotnet add package** command will add the **Microsoft.EntityFrameworkCore.SqlServer** package from **NuGet**. For more information, go to: Microsoft.EntityFrameworkCore.SqlServer.

6. Using the same terminal, import version 3.4.1 of **Microsoft.Azure.Cosmos** from NuGet:

```
dotnet add package Microsoft.Azure.Cosmos --version 3.4.1
```

Note: The **dotnet add package** command will add the **Microsoft.Azure.Cosmos** package from **NuGet**. For more information, go to: Microsoft.Azure.Cosmos.

7. Using the same terminal, build the .NET console application:

```
dotnet build
```

8. Close the current terminal.

Task 2: Create a .NET class

1. Open the **AdventureWorks.Migrate/Program.cs** file in Visual Studio Code.
2. Delete all existing code in the **Program.cs** file.
3. Add the following **using** directives for libraries that will be referenced by the application:

```
using AdventureWorks.Context;
using AdventureWorks.Models;
using Microsoft.Azure.Cosmos;
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

4. Create a new **Program** class with two constant string properties and an asynchronous **Main** entry point method:

```
public class Program
{
    private const string sqlDBConnectionString = "";
    private const string cosmosDBConnectionString = "";

    public static async Task Main(string[] args)
    {
    }

}
```

5. Update the **sqlDBConnectionString** string constant by setting its value to the **ADO.NET (SQL Authentication) connection string** of the SQL database that you recorded earlier in this lab.

Note: It's important that you use your updated connection string here. The original connection string copied from the portal won't have the username and password necessary to connect to the SQL database.

6. Update the **cosmosDBConnectionString** string constant by setting its value to the **PRIMARY CONNECTION STRING** of the Azure Cosmos DB account that you recorded earlier in this lab.

Task 3: Get SQL database records by using Entity Framework

1. Within the **Main** method, add the following blocks of code to export all model and product records from SQL Database to local memory:

```
await Console.Out.WriteLineAsync("Start Migration");

using AdventureWorksSqlContext context = new AdventureWorksSqlContext(sqlDBConnectionString);

List<Model> items = await context.Models
    .Include(m => m.Products)
    .ToListAsync<Model>();

await Console.Out.WriteLineAsync($"Total Azure SQL DB Records: {items.Count}");
```

2. Save the **Program.cs** file.
3. Using a terminal, change your context to the **AdventureWorks.Migrate** folder:

```
cd .\AdventureWorks.Migrate\
```

4. Using the same terminal, build the .NET console application project:

```
dotnet build
```

Note: If there are any build errors, review the **Program.cs** file in the **Allfiles (F):\Allfiles\Labs\04\Solution\AdventureWorks\AdventureWorks.Migrate** folder.

5. Close the current terminal.

Task 4: Insert items into Azure Cosmos DB

1. Still within the **Main** method, add the following blocks of code to import the in-memory model and product data as documents to Azure Cosmos DB:

```
using CosmosClient client = new CosmosClient(cosmosDBConnectionString);

Database database = await client.CreateDatabaseIfNotExistsAsync("Retail");

Container container = await database.CreateContainerIfNotExistsAsync("Online",
    partitionKeyPath: $"/{nameof(Model.Category)}",
    throughput: 1000
);

int count = 0;
foreach (var item in items)
{
    ItemResponse<Model> document = await container.UpsertItemAsync<Model>(item);
    await Console.Out.WriteLineAsync($"Upserted document #{++count:000} [Activity Id: {document.ActivityId}]");
}

await Console.Out.WriteLineAsync($"Total Azure Cosmos DB Documents: {count}");
```

2. Save the **Program.cs** file.
3. Using a terminal, change your context to the **AdventureWorks.Migrate** folder:

```
cd .\AdventureWorks.Migrate\
```

4. Using the same terminal, build the .NET console application project:

```
dotnet build
```

Note: If there are any build errors, review the **Program.cs** file in the **Allfiles (F):\Allfiles\Labs\04\Solution\AdventureWorks\AdventureWorks.Migrate** folder.

Task 5: Perform a migration

1. Using the same terminal, run the .NET console application project:

```
dotnet run
```

Note: The **dotnet run** command will start the console application.

2. Observe the various data that prints to the screen, including initial SQL record count, individual upsert activity identifiers, and final Azure Cosmos DB document count.
3. Close the current terminal.

Task 6: Validate the migration

1. Return to the **Microsoft Edge** browser window with the Azure portal.
2. Go to the blade for the **polycosmos*[yourname]*** Azure Cosmos DB account that you created earlier in this lab.
3. Open the Data Explorer pane.
4. Create a new **SQL query** tab within the context of the **Retail** database and **Online** container.
5. Run the following query, and then observe the results:

```
SELECT * FROM models
```

6. Run the following query, and then observe the results:

```
SELECT VALUE COUNT(1) FROM models
```

Review

In this exercise, you used Entity Framework and the .NET SDK for Azure Cosmos DB to migrate data from SQL Database to Azure Cosmos DB.

Exercise 5: Accessing Azure Cosmos DB by using .NET

Task 1: Update library with the Cosmos SDK and references

1. Using a terminal, change your context to the **AdventureWorks.Context** folder:

```
cd .\AdventureWorks.Context\
```

2. Using the same terminal, import **Microsoft.Azure.Cosmos** from NuGet:

```
dotnet add package Microsoft.Azure.Cosmos --version 3.4.1
```

> **Note**: The **dotnet add package** command will add the **Microsoft.Azure.Cosmos** package from **NuGet**. For more information, go to: Microsoft.Azure.Cosmos.

3. Using the same terminal, build the ASP.NET web application project:

```
dotnet build
```

4. Close the current terminal.

Task 2: Write .NET code to connect to Azure Cosmos DB

1. Create a new **AdventureWorks.Context/AdventureWorksCosmosContext.cs** file in Visual Studio Code.
2. Add the following **using** directives for libraries that will be referenced by the application:

```
using AdventureWorks.Models;
using Microsoft.Azure.Cosmos;
using Microsoft.Azure.Cosmos.Linq;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
```

3. Enter the following code to add an **AdventureWorks.Context** namespace block:

```
namespace AdventureWorks.Context
{
}
```

4. Create a new **AdventureWorksCosmosContext** class that implements the **IAdventureWorksProductContext** interface with a single read-only *Container* variable:

```
public class AdventureWorksCosmosContext : IAdventureWorksProductContext
{
    private readonly Container _container;
}
```

5. Within the **AdventureWorksCosmosContext** class, add a new constructor that creates a new instance of the **CosmosClient** class, and then obtain both a **Database** and **Container** instance from the client:

```
public AdventureWorksCosmosContext(string connectionString, string database = "Retail", string container = "Online")
{
    _container = new CosmosClient(connectionString)
        .GetDatabase(database)
        .GetContainer(container);
}
```

6. Within the **AdventureWorksCosmosContext** class, add a new **FindModelAsync** method that creates a LINQ query, transforms it into an iterator, iterates over the result set, and then returns the single item in the result set:

```
public async Task<Model> FindModelAsync(Guid id)
{
    var iterator = _container.GetItemLinqQueryable<Model>()
        .Where(m => m.id == id)
        .ToFeedIterator<Model>();

    List&lt;Model&gt; matches = new List&lt;Model&gt;();
    while (iterator.HasMoreResults)
    {
        var next = await iterator.ReadNextAsync();
        matches.AddRange(next);
    }

    return matches.SingleOrDefault();

}
```

7. Within the **AdventureWorksCosmosContext** class, add a new **GetModelsAsync** method that runs an SQL query, gets the query result iterator, iterates over the result set, and then returns the union of all results:

```
public async Task<List<Model>> GetModelsAsync()
{
    string query = $@"SELECT * FROM items";

    var iterator = _container.GetItemQueryIterator&lt;Model&gt;(query);

    List&lt;Model&gt; matches = new List&lt;Model&gt;();
    while (iterator.HasMoreResults)
    {
        var next = await iterator.ReadNextAsync();
        matches.AddRange(next);
    }

    return matches;

}
```

8. Within the **AdventureWorksCosmosContext** class, add a new **FindProductAsync** method that runs an SQL query, gets the query result iterator, iterates over the result set, and then returns the single item in the result set:

```
public async Task<Product> FindProductAsync(Guid id)
{
    string query = $@"SELECT VALUE products
                        FROM models
                        JOIN products in models.Products
                        WHERE products.id = '{id}'";

    var iterator = _container.GetItemQueryIterator&lt;Product&gt;(query);

    List&lt;Product&gt; matches = new List&lt;Product&gt;();
    while (iterator.HasMoreResults)
    {
        var next = await iterator.ReadNextAsync();
        matches.AddRange(next);
    }

    return matches.SingleOrDefault();

}
```

9. Save the **AdventureWorksCosmosContext.cs** file.
10. Using a terminal, change your context to the **AdventureWorks.Context** folder:

```
cd .\AdventureWorks.Context\
```

11. Using the same terminal, build the ASP.NET web application project:

```
dotnet build
```

Note: If there are any build errors, review the **AdventureWorksCosmosContext.cs** file in the **Allfiles (F):\Allfiles\Labs\04\Solution\AdventureWorks\AdventureWorks.Context**folder.

12. Close the current terminal.

Task 3: Update the Azure Cosmos DB connection string

1. Open the **AdventureWorks.Web/appsettings.json** file in Visual Studio Code.
2. Replace the value of the *ConnectionStrings.AdventureWorksCosmosContext* property with the **PRIMARY CONNECTION STRING** of the Azure Cosmos DB account that you recorded earlier in this lab.
3. Save the **appsettings.json** file.

Task 4: Update .NET application startup logic

1. Open the **AdventureWorks.Web/Startup.cs** file in Visual Studio Code.
2. In the **Startup** class, find the existing **ConfigureProductService** method.

   Note: The current product service uses SQL as its database.

3. Replace the **ConfigureProductService** method with the following code:

```
public void ConfigureProductService(IServiceCollection services)
{
    services.AddScoped<IAdventureWorksProductContext, AdventureWorksCosmosContext>(provider =>
            new AdventureWorksCosmosContext(
                _configuration.GetConnectionString(nameof(AdventureWorksCosmosContext))
            )
    );
}
```

4. Save the **Startup.cs** file.

Task 5: Validate that the .NET application successfully connects to Azure Cosmos DB

1. Using a terminal, change your context to the **AdventureWorks.Web** folder:

```
cd .\AdventureWorks.Web\
```

2. Using the same terminal, run the ASP.NET web application project:

```
dotnet run
```

   Note: The **dotnet run** command will automatically build any changes to the project and then start the web application without a debugger attached. The command will output the URL of the running application and any assigned ports.

3. Open the Microsoft Edge browser.
4. In the open browser window, browse to the web application that's hosted at **localhost** on port **5000**.

   Note: The URL is http://localhost:5000.

5. In the web application, observe the list of models displayed from the front page.
6. Find the **Touring-1000** model, and then select **View Details**.
7. From the **Touring-1000** product detail page, perform the following actions:
   a. In the **Select options** list, select **Touring-1000 Yellow, 50, $2,384.07**.
   b. Find **Add to Cart**.
8. Close the browser window displaying your web application.
9. Return to the **Visual Studio Code** window.
10. Close the current terminal.

Review

In this exercise, you wrote C# code to query an Azure Cosmos DB collection by using the .NET SDK.

Exercise 6: Clean up your subscription

Task 1: Open Azure Cloud Shell

1. In the Azure portal, select the **Cloud Shell** icon to open a new shell instance.
2. If Cloud Shell isn't already configured, configure the shell for Bash by using the default settings.

Task 2: Delete resource groups

1. Enter the following command, and then select Enter to delete the **PolyglotData** resource group:

```
az group delete --name PolyglotData --no-wait --yes
```

2. Close the Cloud Shell pane in the portal.

Task 3: Close the active applications

1. Close the currently running Microsoft Edge application.
2. Close the currently running Visual Studio Code application.

Review

In this exercise, you cleaned up your subscription by removing the resource groups used in this lab.

Congratulations!

You have successfully completed this exercise. Click **End** to advance to the next exercise.