# Lab: Constructing a polyglot data solution

## Microsoft Azure user interface

Given the dynamic nature of Microsoft cloud tools, you might experience Azure user interface (UI) changes following the development of this training content. These changes might cause the lab instructions and steps to not match up correctly.

Microsoft updates this training course as soon as the community brings needed changes to our attention. However, because cloud updates occur frequently, you might encounter UI changes before this training content is updated. **If this occurs, adapt to the changes and work through them in the labs as needed.**

## Instructions

### Before you start

#### Sign in to lab virtual machine

* Sign in to your **Windows 10** virtual machine using the following credentials:
  + **Username**: Admin
  + **Password**: Pa55w.rd

**Note**: Lab virtual machine sign in instructions will be provided to you by your instructor.

#### Review installed applications

* Observe the taskbar located at the bottom of your **Windows 10** desktop. The taskbar contains the icons for the applications you will use in this lab:
  + Microsoft Edge
  + File Explorer
  + Visual Studio Code

#### Download the lab files

1. On the taskbar, select the **Windows PowerShell** icon.
2. In the PowerShell command prompt, change the current working directory to the **Allfiles (F):\** path:

* cd F:

1. Within the command prompt, enter the following command and press Enter to Clone the **microsoftlearning/AZ-203-DevelopingSolutionsForAzure** project hosted on GitHub into the **Labfiles** directory:

* git clone --depth 1 --no-checkout https://github.com/microsoftlearning/AZ-203-DevelopingSolutionsForMicrosoftAzure .

1. Within the command prompt, enter the following command and press **Enter** to check out the lab files necessary to complete the **AZ-201.02** lab:

* git checkout master -- Allfiles/\*

1. Close the currently running **Windows PowerShell** command prompt application.

### Exercise 1: Creating database resources in Azure

#### Task 1: Open the Azure portal

1. On the taskbar, select the **Microsoft Edge** icon.
2. In the open browser window, navigate to the **Azure portal** ([portal.azure.com](https://portal.azure.com)).
3. Enter the **email address** for your Microsoft account.
4. Select **Next**.
5. Enter the **password** for your Microsoft account.
6. Select **Sign in**.

* **Note**: If this is your first time signing in to the **Azure Portal**, a dialog box will appear offering a tour of the portal. Select **Get Started** to skip the tour and begin using the portal.

#### Task 2: Create a SQL Database resource

1. In the navigation pane on the left side of the Azure portal, select **All services**.
2. In the **All services** blade, select **SQL servers**.
3. In the **SQL servers** blade, view your list of SQL server instances.
4. At the top of the **SQL servers** blade, select **+ Add**.
5. In the **SQL Server (logical server only)** blade, perform the following actions:
   1. In the **Server name** field, enter the value **polysqlsrvr[your name in lowercase]**.
   2. In the **Server admin login** field, enter the value **testuser**.
   3. In the **Password** field, enter the value **TestPa$$w0rd**.
   4. In the **Confirm password** field, enter the value **TestPa$$w0rd** again.
   5. Leave the **Subscription** drop-down list set to its default value.
   6. In the **Resource group** section, select **Create new**, enter **PolyglotData**, and then select **OK**.
   7. In the **Location** drop-down list, select the **(US) East US** option.
   8. In the **Allow Azure Services to access server** section, ensure that the checkbox is selected.
   9. In the **Advanced Data Security** section, select the **Not now** option.
   10. Select **Create**.
6. Wait for the creation task to complete before you move forward with this lab.
7. In the navigation pane on the left side of the Azure portal, select **All services**.
8. In the **All services** blade, select **SQL databases**.
9. In the **SQL databases** blade, view your list of SQL database instances.
10. At the top of the **SQL databases** blade, select **+ Add**.
11. In the **Create SQL Database** blade, observe the tabs at the top of the blade, such as **Basics**, **Additional settings** and **Tags**.

* **Note**: Each tab represents a step in the workflow to create a new **Azure SQL database**. At any time, you can select **Review + create** to skip the remaining tabs.

1. In the **Basics** tab, perform the following actions:
   1. Leave the **Subscription** text box set to its default value.
   2. In the **Resource group** section, select **PolyglotData** from the list.
   3. In the **Databasename** text box, enter **polysqldb**.
   4. In the **Server** field, select the **polysqlsrvr[your name in lowercase]** option.
   5. In the **Want to use SQL elastic pool** section, select the **No** option.
   6. Leave the **Compute + storage** option set to its default value.
   7. Select **Review + Create**.
2. In the **Review + Create** tab, review the options that you selected during the previous steps.
3. Select **Create** to create the SQL Database by using your specified configuration.
4. Wait for the creation task to complete before you move forward with this lab.
5. In the navigation pane on the left side of the Azure portal, select **All services**.
6. In the **All services** blade, select **SQL Database**.
7. In the **SQL Database** blade, select the SQL database instance named **polysqldb**.
8. In the **SQL database** blade, locate the **Settings** section on the left side of the blade and select the **Connection strings** link.
9. In the **Connection strings** pane, copy the value of the **ADO.NET** connection string. Be sure to replace the placeholder values for *{your\_username}* and *{your\_password}* with the values testuserand TestPa$$w0rd, respectively.

* **Note**: For example, if your copied connection string is
* Server=tcp:polysqlsrvrstudent.database.windows.net,1433;Initial Catalog=polysqldb;Persist Security Info=False;User ID={your\_username};Password={your\_password};MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;,
* your updated connection string would be
* Server=tcp:polysqlsrvrstudent.database.windows.net,1433;Initial Catalog=polysqldb;Persist Security Info=False;User ID=testuser;Password=TestPa$$w0rd;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;

1. In the navigation pane on the left side of the Azure portal, select **All services**.
2. In the **All services** blade, select **SQL servers**.
3. In the **SQL servers** blade, select the SQL server instance that has the prefix **polysqlsrvr**.
4. In the **SQL server** blade, locate the **Security** section on the left side of the blade and select the **Firewalls and virtual networks** link.
5. In the **Firewalls and virtual networks** pane, select **Add client IP** to add your virtual machine’s IP address to the list of allowed IP address ranges.
6. At the top of the blade, select **Save**.

* **Note**: It might take a few minutes for the firewall changes to get updated on the server.

1. After the save operation is complete, select **OK** to dismiss the confirmation dialog.

#### Task 3: Create an Azure Cosmos DB account resource

1. In the navigation pane on the left side of the Azure portal, select **All services**.
2. In the **All services** blade, select **Azure Cosmos DB**.
3. In the **Azure Cosmos DB** blade, view your list of Azure Cosmos DB instances.
4. At the top of the **Azure Cosmos DB** blade, select **+ Add**.
5. In the **Create Azure Cosmos DB Account** blade, observe the tabs at the top of the blade, such as **Basics**, **Network** and **Tags**.

* **Note**: Each tab represents a step in the workflow to create a new **Azure Cosmos DB account**. At any time, you can select **Review + create** to skip the remaining tabs.

1. In the **Basics** tab, perform the following actions:
   1. Leave the **Subscription** text box set to its default value.
   2. In the **Resource group** section, select **PolyglotData** from the list.
   3. In the **AccountName** text box, enter **polycosmos[your name in lowercase]**.
   4. In the **API** drop-down list, select the **Core (SQL)** option.
   5. In the **Location** drop-down list, select the **East US** region.
   6. In the **Geo-Redundancy** section, select the **Disable** option.
   7. In the **Multi-region Writes** section, select the **Disable** option.
   8. Select **Review + Create**.
2. In the **Review + Create** tab, review the options that you selected during the previous steps.
3. Select **Create** to create the Azure Cosmos DB account by using your specified configuration.
4. Wait for the creation task to complete before you move forward with this lab.
5. In the navigation pane on the left side of the Azure portal, select **All services**.
6. In the **All services** blade, select **Azure Cosmos DB**.
7. In the **Azure Cosmos DB** blade, select the Azure Cosmos DB account instance that contains the prefix **polycosmos**.
8. In the **Azure Cosmos DB account** blade, locate the **Settings** section on the left side of the blade and select the **Keys** link.
9. In the **Keys** pane, record the values in the **URI** and **PRIMARY KEY** fields. You will use these values later in this lab.

#### Task 4: Create an Azure Storage account resource

1. In the navigation pane on the left side of the Azure portal, select **All services**.
2. In the **All services** blade, select **Storage Accounts**.
3. In the **Storage accounts** blade, view your list of Storage instances.
4. At the top of the **Storage accounts** blade, select **+ Add**.
5. In the **Create storage account** blade, observe the tabs at the top of the blade, such as **Basics**, **Advanced** and **Tags**.

* **Note**: Each tab represents a step in the workflow to create a new **Azure Storage Account**. At any time, you can select **Review + create** to skip the remaining tabs.

1. In the **Basics** tab, perform the following actions:
   1. Leave the **Subscription** text box set to its default value.
   2. In the **Resource group** section, select **PolyglotData** from the list.
   3. In the **Storage account name** text box, enter **polystor[your name in lowercase]**.
   4. In the **Location** drop-down list, select the **East US** region.
   5. In the **Performance** section, select **Standard**.
   6. In the **Account kind** drop-down list, select **StorageV2 (general purpose v2)***.*
   7. In the **Replication** drop-down list, select **Locally-redundant storage (LRS)**.
   8. In the **Access tier (default)** section, ensure that **Hot**is selected.
   9. Select **Review + Create**.
2. In the **Review + Create** tab, review the options that you selected during the previous steps.
3. Select **Create** to create the storage account by using your specified configuration.
4. Wait for the creation task to complete before you move forward with this lab.
5. In the navigation pane on the left side of the Azure portal, select **All services**.
6. In the **All services** blade, select **Storage Accounts**.
7. In the **Storage accounts** blade, select the Azure Storage account instance that has the prefix **polystor**.
8. In the **Storage account** blade, locate the **Settings** section on the left side of the blade and select the **Access keys** link.
9. In the **Access keys** blade, select any one of the keys and record the value in the **Connection string** text box. You will use this value later in this lab.

#### Review

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

### Exercise 2: Open and configure an ASP.NET Core web application

#### Task 1: Open the web application

1. On the **Start** screen, select the **Visual Studio Code** tile.
2. On the **File** menu, select **Open Folder**.
3. In the File Explorer pane that opens, go to **Allfiles (F):\Allfiles\03\Starter**, and then select **Select Folder**.

#### Task 2: Update application settings

1. In the **Explorer** pane of the Visual Studio Code window, expand the **Contoso.Events.Web** project.
2. In the **Explorer** pane, double-select **appsettings.json**.
3. In the JSON object, in line 13, locate the **ConnectionStrings.EventsContextConnectionString** path. Observe that the current value is empty:

* "ConnectionStrings": {  
  "EventsContextConnectionString": ""  
  },

1. Update the value of the **EventsContextConnectionString** property by setting its value to the **connection string** of the **SQL database** that you recorded earlier in this lab.
2. In the JSON object, in line 9, locate the **CosmosSettings.EndpointUrl** path. Observe that the current value is empty:

* "CosmosSettings": {  
  "DatabaseId": "EventsDatabase ",  
  "CollectionId": "RegistrationCollection",  
  "EndpointUrl": "",  
  "AuthorizationKey": ""  
  },

1. Update the value of the **EndpointUrl** property by setting its value to the **Endpoint Uri** of the **Azure Cosmos DB account that** you recorded earlier in this lab.
2. In the JSON object, in line 10, locate the **CosmosSettings.AuthorizationKey** path. Observe that the current value is empty:

* "CosmosSettings": {  
  "DatabaseId": "EventsDatabase ",  
  "CollectionId": "RegistrationCollection",  
  "EndpointUrl": "",  
  "AuthorizationKey": ""  
  },

1. Update the value of the **AuthorizationKey** property by setting its value to the **Key** of the **Azure Cosmos DB account** that you recorded earlier in this lab.
2. Save the **appsettings.json** file.
3. In the **Explorer** pane of the Visual Studio Code window, expand the **Contoso.Events.Worker** project.
4. In the **Explorer** pane, double-select **local.settings.json**.
5. In the JSON object, in line 4, locate the **AzureWebJobsStorage** path. Observe that the current value is empty:

* "AzureWebJobsStorage": "",

1. Update the value of the **AzureWebJobsStorage** property by setting its value to the **connection string** of the **storage account** that you recorded earlier in this lab.
2. In the JSON object, in line 5, locate the **AzureWebJobsDashboard** path. Observe that the current value is empty:

* "AzureWebJobsDashboard": "",

1. Update the value of the **AzureWebJobsDashboard** property by setting its value to the **connection string** of the **storage account** that you recorded earlier in this lab.
2. In the JSON object, in line 6, locate the **EventsContextConnectionString** path. Observe that the current value is empty:

* "EventsContextConnectionString": "",

1. Update the value of the **EventsContextConnectionString** property by setting its value to the **connection String** of the **SQL database** you recorded earlier in this lab.
2. In the JSON object, in line 7, locate the **CosmosEndpointUrl** path. Observe that the current value is empty:

* "CosmosEndpointUrl": "",

1. Update the value of the **CosmosEndpointUrl** property by setting its value to the **Endpoint Uri** of the **Azure Cosmos DB account** that you recorded earlier in this lab.
2. In the JSON object, in line 8, locate the **CosmosAuthorizationKey** path. Observe that the current value is empty:

* "CosmosAuthorizationKey": "",

1. Update the value of the **CosmosAuthorizationKey** property by setting its value to the **Key** of the **Azure Cosmos DB account** that you recorded earlier in this lab.
2. Save the **local.settings.json** file.

#### Review

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

### Exercise 3: Authoring Entity Framework code to connect to Azure SQL Database

#### Task 1: Configure database initialization logic

1. In the Explorer pane of the Visual Studio Code window, expand the **Contoso.Events.Data** project.
2. In the Explorerpane, double-select **ContextInitializer.cs**.
3. Locate the **InitializeAsync** method:

* public async Task InitializeAsync(EventsContext eventsContext)

1. Within the **InitializeAsync** method, add the following code line to ensure that the database is created:

* await eventsContext.Database.EnsureCreatedAsync();

1. Add the following code block to create a conditional **if** block that only executes the code within the block if there are no events in the database:

* if (!await eventsContext.Events.AnyAsync())  
  {  
  }

1. Within the newly created **if** block, add the following code line to create a new instance of the **Event** class:

* Event eventItem = new Event();

1. Within the **if** block, add the following code block to set various properties of the new **Event** class instance:

* eventItem.EventKey = "FY17SepGeneralConference";  
  eventItem.StartTime = DateTime.Today;  
  eventItem.EndTime = DateTime.Today.AddDays(3d);  
  eventItem.Title = "FY17 September Technical Conference";  
  eventItem.Description = "Sed in euismod mi.";  
  eventItem.RegistrationCount = 1;

1. Within the **if** block, add the following code line to add the new **Event** class instance to the **Events** property of type **DbSet<Event>**:

* eventsContext.Events.Add(eventItem);

1. Outside of and after the **if**block, add the following code line to save the changes to the database context:

* await eventsContext.SaveChangesAsync();

1. Your **InitializeAsync** method should now look like this:

* public async Task InitializeAsync(EventsContext eventsContext)  
  {  
  await eventsContext.Database.EnsureCreatedAsync();  
    
  if (!await eventsContext.Events.AnyAsync())  
  {  
  Event eventItem = new Event();  
  eventItem.EventKey = "FY17SepGeneralConference";  
  eventItem.StartTime = DateTime.Today;  
  eventItem.EndTime = DateTime.Today.AddDays(3d);  
  eventItem.Title = "FY17 September Technical Conference";  
  eventItem.Description = "Sed in euismod mi.";  
  eventItem.RegistrationCount = 1;  
  eventsContext.Events.Add(eventItem);  
  }  
    
  await eventsContext.SaveChangesAsync();  
  }

1. **Save** the **ContextInitializer.cs** file.

#### Task 2: Update the database initialization

1. In the Explorer pane of the Visual Studio Code window, expand the **Contoso.Events.Data** project.
2. In the **Explorer** pane, double-select **ContextInitializer.cs**.
3. Locate the **InitializeAsync** method:

* public async Task InitializeAsync(EventsContext eventsContext)

1. Replace the method with the following method implementation:

* public async Task InitializeAsync(EventsContext eventsContext)  
  {  
  await eventsContext.Database.EnsureCreatedAsync();  
    
  if (!await eventsContext.Events.AnyAsync())  
  {  
  await eventsContext.Events.AddRangeAsync(  
  new List<Event>()   
  {  
  new Event { EventKey = "GeneralConferenceAlpha", StartTime = DateTime.Today, EndTime = DateTime.Today.AddDays(5d), Title = "First General Conference", Description = "Sed in euismod mi.", RegistrationCount = 15 },  
  new Event { EventKey = "GeneralConferenceBravo", StartTime = DateTime.Today.AddDays(10d), EndTime = DateTime.Today.AddDays(15d), Title = "Second General Conference", Description = "Sed in euismod mi.", RegistrationCount = 20 },  
  new Event { EventKey = "GeneralConferenceCharlie", StartTime = DateTime.Today.AddDays(20d), EndTime = DateTime.Today.AddDays(25d), Title = "Third General Conference", Description = "Sed in euismod mi.", RegistrationCount = 5 },  
  new Event { EventKey = "GeneralConferenceDelta", StartTime = DateTime.Today.AddDays(30d), EndTime = DateTime.Today.AddDays(35d), Title = "Fourth General Conference", Description = "Sed in euismod mi.", RegistrationCount = 25 },  
  new Event { EventKey = "GeneralConferenceEcho", StartTime = DateTime.Today.AddDays(40d), EndTime = DateTime.Today.AddDays(45d), Title = "Fifth General Conference", Description = "Sed in euismod mi.", RegistrationCount = 10 },  
  new Event { EventKey = "GeneralConferenceFoxtrot", StartTime = DateTime.Today.AddDays(50d), EndTime = DateTime.Today.AddDays(55d), Title = "Sixth General Conference", Description = "Sed in euismod mi.", RegistrationCount = 0 }  
  }  
  );  
    
  await eventsContext.SaveChangesAsync();  
  }  
  }

1. **Save** the **ContextInitializer.cs** file.

#### Task 3: Write Entity Framework queries in the ASP.NET MVC controllers

1. In the Explorerpane of the Visual Studio Code window, expand the **Contoso.Events.Web** project.
2. In the Explorer pane, expand the **Controllers** folder.
3. In the Explorerpane, double-select **HomeController.cs**.
4. Locate the **Index** method:

* public IActionResult Index([FromServices] EventsContext eventsContext, [FromServices] IOptions<ApplicationSettings> appSettings)

1. Within the **Index** method, locate the following code line:

* var upcomingEvents = Enumerable.Empty<Event>();

1. Replace that code line with the following code block to query the **Events** table, order the results by the **StartTime** property and then retrieve (take) a subset of results based on an application setting:

* var upcomingEvents = eventsContext.Events  
  .Where(e => e.StartTime >= DateTime.Today)  
  .OrderBy(e => e.StartTime)  
  .Take(appSettings.Value.LatestEventCount);

1. **Save** the **HomeController.cs** file.
2. In the Explorerpane, double-select **EventsController.cs**.
3. Locate the **Index** method:

* public IActionResult Index([FromServices] EventsContext eventsContext, [FromServices] IOptions<ApplicationSettings> appSettings, int? page)

1. Within the **Index** method, locate the following code line:

* var pagedEvents = Enumerable.Empty<Event>();

1. Replace that code line with the following code block to query the **Events** table, and use the **Skip** and **Take** methods to create a page of results based on the current page number:

* int currentPage = page ?? 1;  
  int totalRows = eventsContext.Events.Count();  
  int pageSize = appSettings.Value.GridPageSize;  
  var pagedEvents = eventsContext.Events  
  .OrderByDescending(e => e.StartTime)  
  .Skip(pageSize \* (currentPage - 1))  
  .Take(pageSize);

1. Within the **Index** method, locate the following code block:

* EventsGridViewModel viewModel = new EventsGridViewModel  
  {  
  CurrentPage = 0,  
  PageSize = 0,  
  TotalRows = 0,  
  Events = pagedEvents  
  };

1. Replace that code block with the following code block to set various properties of the **EventsGridViewModel** class instance:

* EventsGridViewModel viewModel = new EventsGridViewModel  
  {  
  CurrentPage = currentPage,  
  PageSize = pageSize,  
  TotalRows = totalRows,  
  Events = pagedEvents  
  };

1. Locate the **Detail** method:

* public IActionResult Detail([FromServices] EventsContext eventsContext, string key)

1. Within the **Detail** method, locate the following code line:

* var matchedEvent = default(Event);

1. Replace that code line with the following code block to query the **Events** table for a single record that matches the **EventKey** property:

* var matchedEvent = eventsContext.Events  
  .SingleOrDefault(e => e.EventKey == key);

1. **Save** the **EventsController.cs** file.

#### Review

In this exercise, you wrote C# code to connect to an Azure SQL database by using Entity Framework.

### Exercise 4: Authoring Cosmos DB Client Library code to connect to Azure Cosmos DB

#### Task 1: Retrieve registrant names in the Azure Functions project

1. In the Explorer pane of the Visual Studio Code window, expand the **Contoso.Events.Worker** project and double-select the **ProcessDocuments.cs** file.
2. In the code editor tab for the **ProcessDocuments.cs** file, locate the **ProcessDocuments** class:

* public static class ProcessDocuments

1. Within the **ProcessDocuments** class, add a new line of code at line 19 to create a new static instance of the **RegistrationContext** class:

* private static RegistrationContext \_registrationsContext = \_connection.GetCosmosContext();

1. Locate the **ProcessHttpRequestMessage** method:

* private static async Task<List<string>> ProcessHttpRequestMessage(string eventKey)

1. Within the **ProcessHttpRequestMessage** method, add a new line of code at line 40 to configure the connection to Azure Cosmos DB by using the **ConfigureConnectionAsync** method of the **RegistrationContext** class:

* await \_registrationsContext.ConfigureConnectionAsync();

1. Locate the line of code at line 41 that stores an empty collection of string values in the **registrants** variable:

* List<string> registrants = new List<string>();

1. Replace the line of code at line 41 with a new line of code that invokes the **GetRegistrantsForEvent** method of the **RegistrationContext** class:

* List<string> registrants = await \_registrationsContext.GetRegistrantsForEvent(eventKey);

1. Save the **ProcessDocuments.cs** file.

#### Task 2: Implement a RegistrationContext class

1. In the **Explorer** pane of the Visual Studio Code window, expand the **Contoso.Events.Data** project and double-select the **RegistrationContext.cs** file.
2. In the code editor tab for the **RegistrationContext.cs** file, locate the **RegistrationContext** class:

* public class RegistrationContext

1. Within the **RegistrationContext** class, add a new line of code to create a new property of type **Database**:

* protected Database Database { get; set; }

1. Within the **RegistrationContext** class, add a new line of code to create a new property of type **DocumentCollection**:

* protected DocumentCollection Collection { get; set; }

1. Within the **RegistrationContext** class, add a new line of code to create a new property of type **DocumentClient**:

* protected DocumentClient Client { get; set; }

1. Within the **RegistrationContext** class, locate the existing **RegistrationContext** constructor:

* public RegistrationContext(IOptions<CosmosSettings> cosmosSettings)  
  {  
  CosmosSettings = cosmosSettings.Value;  
  }

1. Within the constructor, add a new line of code to create a new **DocumentClient** instance and save it to the *Client* property:

* Client = new DocumentClient(new Uri(CosmosSettings.EndpointUrl), CosmosSettings.AuthorizationKey);

1. Within the **RegistrationContext** class, locate the **ConfigureConnectionAsync** method and delete any existing code within the method:

* public async Task ConfigureConnectionAsync()  
  {  
    
  }

1. Within the **ConfigureConnectionAsync** method, add a new line of code to create a new **Database** instance and save it to the **Database** property:

* Database = await Client.CreateDatabaseIfNotExistsAsync(new Database { Id = CosmosSettings.DatabaseId });

1. Add a new line of code to create a new **DocumentCollection** instance and save it to the **Collection** property:

* Collection = await Client.CreateDocumentCollectionIfNotExistsAsync(Database.SelfLink, new DocumentCollection { Id = CosmosSettings.CollectionId });

1. Within the **RegistrationContext** class, locate the **GetRegistrantCountAsync** method and delete any existing code within the method:

* public async Task<int> GetRegistrantCountAsync()  
  {  
    
  }

1. Within the **GetRegistrantCountAsync** method, add a new line of code to create a new instance of the **FeedOptions** class with its **EnableCrossPartitionQuery** property set to a value of true:

* FeedOptions options = new FeedOptions { EnableCrossPartitionQuery = true };

1. Add a new line of code to create a query that gets the count of registrants and returns a collection of integer values:

* IDocumentQuery<int> query = Client.CreateDocumentQuery<int>(Collection.SelfLink, "SELECT VALUE COUNT(1) FROM registrants", options).AsDocumentQuery();

1. Add a new block of code to create an integer variable named ***count***, a while loop that iterates over the **HasMoreResults** property of the **IDocumentQuery<>** class and returns the final value of the ***count***variable:

* int count = 0;  
  while (query.HasMoreResults)  
  {  
    
  }  
  return count;

1. Within the while loop, add a line of code to invoke the **ExecuteNextAsync<>** method of the **IDocumentQuery<>** class and store its result in a variable named ***results***of type **FeedResponse<>**:

* FeedResponse<int> results = await query.ExecuteNextAsync<int>();

1. Within the while loop, add a code line to increment the ***count*** variable by the result of invoking the **Sum** LINQ method on the collection stored in the ***results*** variable:

* count += results.Sum();

1. Within the **RegistrationContext** class, locate the **GetRegistrantsForEvent** method and delete any existing code within the method:

* public async Task<List<string>> GetRegistrantsForEvent(string eventKey)  
  {  
    
  }

1. Within the **GetRegistrantsForEvent** method, add a line of code that uses LINQ to get registrants for a specific event by using the value of the ***eventKey***parameter and deserialize those registrants using the **GeneralRegistration** class:

* IDocumentQuery<GeneralRegistration> query = Client.CreateDocumentQuery<GeneralRegistration>(Collection.SelfLink).Where(r => r.EventKey == eventKey).AsDocumentQuery();

1. Add a new block of code to create a **List<>** variable named ***registrants***, a while loop that iterates over the **HasMoreResults** property of the **IDocumentQuery<>** class and returns the final value of the ***registrants*** variable:

* List<string> registrants = new List<string>();  
  while (query.HasMoreResults)  
  {  
    
  }  
  return registrants;

1. Within the while loop, add a line of code to invoke the **ExecuteNextAsync<>** method of the **IDocumentQuery<>** class and store its result in a variable named ***results***of type **FeedResponse<>**:

* FeedResponse<GeneralRegistration> results = await query.ExecuteNextAsync<GeneralRegistration>();

1. Within the while loop, add a line of code that invokes the **Select** LINQ method to project two properties of the **GeneralRegistration** class as a single string and store the resulting collection in a variable named ***resultNames***of type **IEnumerable<>**:

* IEnumerable<string> resultNames = results.Select(r => $"{r.FirstName} {r.LastName}");

1. Within the while loop, add another line of code to append the **registrants** collection with the values in the **resultNames** collection by using the **AddRange** method of the **List<>** class.

* registrants.AddRange(resultNames);

1. Within the **RegistrationContext** class, locate the **UploadEventRegistrationAsync** method and delete any existing code within the method:

* public async Task<string> UploadEventRegistrationAsync(dynamic registration)  
  {  
    
  }

1. Within the **UploadEventRegistrationAsync** method, add a line of code to invoke the **CreateDocumentAsync** method of the **Client** property of type **DocumentClient** and save the result as a variable named ***response*** of type **ResourceResponse<Document>**. This method will upload the document to Azure Cosmos DB:

* ResourceResponse<Document> response = await Client.CreateDocumentAsync(Collection.SelfLink, registration);

1. Add another line of code to use dot notation to access the **Resource** property (of type **Document**) of the ***response*** variable and the **Id** property of the **Document** instance. Return the string **Id** value as the result of the current method:

* return response.Resource.Id;

1. Save the **RegistrationContext.cs** file.

#### Review

In this exercise, you wrote the C# code necessary to access and query documents in Azure Cosmos DB.

### Exercise 5: Authoring Azure SDK code to connect to Azure Storage

#### Task 1: Implement a blob trigger and output for Azure Functions

1. In the **Explorer** pane of the Visual Studio Code window, expand the **Contoso.Events.Worker** project and double-select the **ProcessDocuments.cs** file.
2. In the code editor tab for the **ProcessDocuments.cs** file, locate the **ProcessDocuments** class:

* public static class ProcessDocuments

1. Within the **ProcessDocuments** class, locate the **Run** method:

* public static async Task Run(Stream input, string name, Stream output, TraceWriter log)

1. Update the method signature of the **Run** method by adding a *BlobTrigger*parameter attribute to the *input*parameter specifying to match on any blob in the **signinsheets-pending** container:

* public static async Task Run([BlobTrigger("signinsheets-pending/{name}")] Stream input, string name, Stream output, TraceWriter log)

1. Update the method signature of the **Run** method by adding a *Blob* parameter attribute to the *output* parameter specifying to create a new blob in the **signinsheets** container with the same name as the blob that triggered the Function execution:

* public static async Task Run([BlobTrigger("signinsheets-pending/{name}")] Stream input, string name, [Blob("signinsheets/{name}", FileAccess.Write)] Stream output, TraceWriter log)

1. Within the **Run** method, add a new line of code at line 23 to get the **Event Key** by stripping the file extension from the name of the blob:

* string eventKey = Path.GetFileNameWithoutExtension(name);

1. Add a new line of code at line 24 to create a **using** block by using the **Stream** result from an invocation of the **ProcessStorageMessage** method:

* using (MemoryStream stream = await ProcessStorageMessage(eventKey))  
  {  
    
  }

1. Within the **using** block, add a new line of code to create a new variable of type **byte[]** by invoking the **ToArray** method of the stream:

* byte[] byteArray = stream.ToArray();

1. Within the **using** block, add another line of code to invoke the **WriteAsync** method of the *output* variable passing in various metadata related to the *byteArray*variable:

* 2await output.WriteAsync(byteArray, 0, byteArray.Length);

1. Save the **ProcessDocuments.cs** file.

#### Task 2: Implement a blob upload in BlobContext class

1. In the Explorer pane of the Visual Studio Code window, expand the **Contoso.Events.Data** project and double-select the **BlobContext.cs** file.
2. In the code editor tab for the **BlobContext.cs** file, locate the **BlobContext** class:

* public class BlobContext

1. Within the **BlobContext** class, locate the **UploadBlobAsync** method and delete any existing code within the method:

* public async Task<ICloudBlob> UploadBlobAsync(string blobName, Stream stream)  
  {  
    
  }

1. Within the **UploadBlobAsync** method, add a new line of code to create a new **CloudStorageAccount** class instance:

* CloudStorageAccount account = CloudStorageAccount.Parse(StorageSettings.ConnectionString);

1. Add a new line of code to create a new instance of the **CloudBlobClient** class by using the **CreateCloudBlobClient** method of the **CloudStorageAccount** class:

* CloudBlobClient blobClient = account.CreateCloudBlobClient();

1. Add a new line of code to get a reference to a new or existing container by using the **GetContainerReference** method of the **CloudBlobClient** class:

* CloudBlobContainer container = blobClient.GetContainerReference($"{StorageSettings.ContainerName}-pending");

1. Add a new line of code to invoke the **CreateIfNotExistsAsync** method of the **CloudBlobContainer** class that will create the container if it does not already exist:

* await container.CreateIfNotExistsAsync();

1. Add a new line of code to get a reference to a new or existing blob by using the specified blob name:

* ICloudBlob blob = container.GetBlockBlobReference(blobName);

1. Add a new line of code to take your ***stream*** parameter and revert the stream back to its origin:

* stream.Seek(0, SeekOrigin.Begin);

1. Add a new line of code to upload the ***stream*** parameter’s content to the referenced blob:

* await blob.UploadFromStreamAsync(stream);

1. Add a new line of code to return the updated blob as the result of the method:

* return new DownloadPayload { Stream = stream, ContentType = blob.Properties.ContentType };

1. **Save** the **BlobContext.cs** file.

#### Review

In this exercise, you wrote C# code to manipulate Azure Storage blobs in an Azure Function.

### Exercise 6: Clean up subscription

#### Task 1: Open Azure Cloud Shell

1. At the top of the portal, select the **Cloud Shell** icon to open a new shell instance.
2. In the **Cloud Shell** command prompt at the bottom of the portal, type in the following command and press Enter to list all resource groups in the subscription:

* az group list

1. In the prompt, type the following command and press Enter to view a list of possible commands to delete a resource group:

* az group delete --help

#### Task 2: Delete resource groups

1. In the prompt, type the following command and press Enter to delete the **PolyglotData** resource group:

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

1. Close the **Cloud Shell** pane at the bottom of the portal.

#### Task 3: Close 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.