**Module 1: Introduction to Azure Cosmos DB**

* [Module 1: Introduction to Azure Cosmos DB](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#module-1-introduction-to-azure-cosmos-db)
  + [Lab: Introduction to Azure Cosmos DB](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#lab-introduction-to-azure-cosmos-db)
    - [Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#scenario)
    - [Objectives](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#objectives)
    - [Lab Setup](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#lab-setup)
  + [Exercise 1: Create an Azure Cosmos DB database with a SQL API](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#exercise-1-create-an-azure-cosmos-db-database-with-a-sql-api)
    - [Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#scenario-1)
      * [Task 1: Prepare the environment](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-1-prepare-the-environment)
      * [Task 2: Create a document database using the Cosmos DB SQL API](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-2-create-a-document-database-using-the-cosmos-db-sql-api)
      * [Task 3: Add customer documents to a collection](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-3-add-customer-documents-to-a-collection)
      * [Task 4: Perform SQL queries](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-4-perform-sql-queries)
      * [Task 5: Configure document lifetime](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-5-configure-document-lifetime)
      * [Task 6: Replicate the Cosmos DB Account across regions](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-6-replicate-the-cosmos-db-account-across-regions)
  + [Exercise 2: Migrate data from an existing Mongo DB database to Cosmos DB](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#exercise-2-migrate-data-from-an-existing-mongo-db-database-to-cosmos-db)
    - [Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#scenario-2)
      * [Task 1: Prepare the MongoDB database](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-1-prepare-the-mongodb-database)
      * [Task 2: Run the DocumentDB Data Migration Tool](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-2-run-the-documentdb-data-migration-tool)
      * [Task 3: Import data from the MongoDB database to you Cosmos DB database](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-3-import-data-from-the-mongodb-database-to-you-cosmos-db-database)
  + [Exercise 3: Access data by using the SQL API](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#exercise-3-access-data-by-using-the-sql-api)
    - [Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#scenario-3)
      * [Task 1: Examine the existing .NET application](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-1-examine-the-existing-net-application)
      * [Task 2: Configure the application to connect to Cosmos DB using the SQL API](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-2-configure-the-application-to-connect-to-cosmos-db-using-the-sql-api)
      * [Task 3: Finish the application](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-3-finish-the-application)
      * [Task 4: Test the application](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-4-test-the-application)
  + [Exercise 4: Protect data in a Cosmos DB database](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#exercise-4-protect-data-in-a-cosmos-db-database)
    - [Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#scenario-4)
      * [Task 1: Set permissions for users](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-1-set-permissions-for-users)
      * [Task 2: Configure and run the application](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-2-configure-and-run-the-application)
      * [Task 3: Authenticate users and generate resource tokens](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-3-authenticate-users-and-generate-resource-tokens)
      * [Task 4: Build a client application to test the web API](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-4-build-a-client-application-to-test-the-web-api)
      * [Task 5: Lab close down](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_01.md#task-5-lab-close-down)

**Lab: Introduction to Azure Cosmos DB**

**Scenario**

As a member of the DevOps team in the Adventure Works company, your CTO has asked you to investigate the feasibility of moving your current NoSQL processing from an internally hosted MongoDB database, to a fully managed Azure Cosmos DB instance.

They are keen to understand the complexities of moving the data, and the tools available to protect the data once it is being hosted in Azure.

**Objectives**

After completing this lab, you will be able to:

* Create an Azure Cosmos DB database with a SQL API
* Migrate data from an existing Mongo DB database to Cosmos DB
* Access data by using the SQL API
* Protect data in a Cosmos DB database

**Lab Setup**

* **Estimated time**: 45 minutes
* **Virtual machine**: 20777A-LON-DEV
* **User name**: Administrator
* **Password**: Pa55w.rd

**Exercise 1: Create an Azure Cosmos DB database with a SQL API**

**Scenario**

You decide that the best dataset to move into Azure is the customers data. You will use the Azure portal to creating a document database to hold information about Adventure Works customers. You have decided create the database with the SQL API, as this appears to offer the most functionality and you are more comfortable writing SQL statements than Mongo operations.

The main tasks for this exercise are as follows:

1. Create a document database using the Cosmos DB SQL API.
2. Add customer documents to a collection.
3. Perform SQL queries.
4. Configure a documents lifetime.
5. Replicate the Cosmos DB Account across regions.

**Task 1: Prepare the environment**

1. Ensure that the **MT17B-WS2016-NAT** and **20777A-LON-DEV** virtual machines are running, and then log on to **20777A-LON-DEV** as **Administrator** with the password **Pa55w.rd**.
2. In File Explorer, go to **E:\Labfiles\Lab01**, right-click **Setup.cmd**, and then click **Run as administrator**.
3. Wait until the script has completed, and then press Enter.

**Task 2: Create a document database using the Cosmos DB SQL API**

1. Create a new Cosmos DB account using the SQL API.
2. Create a new database named **SalesData**.
3. Add a new collection named **Customers** to the **SalesData** database. Partition the data using the **/customerid** field.
4. Use fixed storage, with **5000** RU/s throughput. Specify **/customerid** as the unique key.

**Task 3: Add customer documents to a collection**

1. Add the following documents to the Customers collection.
2. {
3. "id": "1",
4. "customerid": 101,
5. "firstname": "Orlando",
6. "lastname": "Gee",
7. "email": "orlando.gee@adventure-works.com",
8. "addresses": [
9. { "addressline1" : "660 Lindbergh", "city" : "Saint Louis", "stateprovince" : "Missouri", "countryregion" : "United States", "postalcode" : "63103" },
10. { "addressline1" : "Ontario Mills", "city" : "Ontario", "stateprovince" : "California", "countryregion" : "United States", "postalcode" : "91764" }
11. ]
12. }
13. {
14. "id": "2",
15. "customerid" : 122,
16. "title" : "Ms.",
17. "firstname" : "Caroline",
18. "middlename" : "A.",
19. "lastname" : "Vicknair",
20. "companyname" : "World of Bikes",
21. "salesperson" : "adventure-works\\jillian0",
22. "addresses": [
23. { "addressline1" : "72540 Blanco Rd.", "addressline2" : "South Side", "city" : "San Antonio", "stateprovince" : "Texas", "countryregion" : "United States", "postalcode" : "78204" }
24. ]
25. }
26. {
27. "id": "3",
28. "customerid": 108,
29. "title" : "Mr.",
30. "firstname" : "Robert",
31. "middlename" : "R.",
32. "lastname" : "Vessa",
33. "suffix" : "Esq.",
34. "companyname" : "Totes & Baskets Company",
35. "salesperson" : "adventure-works\\jillian0",
36. "emailaddress" : "robert13@adventure-works.com",
37. "phone" : "560-555-0171"

}

**Question**: Try and add another document with the customerid field set to 108. What happens?

**Task 4: Perform SQL queries**

1. Run the following queries. Note that the data is returned as an array of JSON docs:
2. SELECT \* FROM c

SELECT c.firstname, c.lastname, c.email FROM c

**Question**: As a document database is schema-less, what is returned in the customers JSON documents that do not have an email property?

SELECT c.Customerid FROM c

**Question**: Why does this query return no data? (**Hint:** field names are case sensitive)

SELECT c.addresses FROM c

**Question**: Explain the format of the documents returned by this query.

SELECT c.customerid, c.city FROM c

**Question**: What might you have been expecting this query to return, and why didn't it work?

**Question**: How would you display the customerid and city for each customer? Is this correct?

SELECT c.customerid, c.addresses.city FROM c

**Task 5: Configure document lifetime**

1. Use the **Scale & Settings** blade try to set the Time to Live (TTL) for the Customers collection to 10 seconds.
2. Save the changes, wait for 10 seconds, and then query the collection again. Note that all the previous documents have been deleted.
3. Reset the TTL for the Customers collection to "Off".

**Task 6: Replicate the Cosmos DB Account across regions**

1. Switch to the Replicate Data Globally blade.
2. Add two new read regions, one close to your existing region, and the other in another continent.
3. Use the Automatic Failover option to set the priorities of the read regions.
4. The read region with the hightest priority (lowest number) will become the write region if the current write region fails.
5. Select Manual Failover. Use this blade to force a failover and specify which region should become the new write region.
6. Switch back to your default region afterwards.

**Result**: At the end of this exercise, you will have:

* Created an Azure Cosmos DB database with a SQL API.
* Added documents to a document database.
* Performed a number of queries against the database.
* Configured documents Time to Live value.
* Replicated your databases data globally.

**Exercise 2: Migrate data from an existing Mongo DB database to Cosmos DB**

**Scenario**

You decide that the best dataset to move into Azure is your customers data. You will use the Azure portal to creating a document database to hold information about Adventure Works customers. You have decided to create the database with the SQL API, as this offers more functionality and you are more comfortable writing SQL statements than Mongo operations.

You then copy your existing companies customer data from your MongoDB into the new Azure Cosmos DB with SQL API.

The main tasks for this exercise are as follows:

1. Prepare the MongoDB database
2. Download the Data Migration tool
3. Import data from the MongoDB database to you Cosmos DB database

**Task 1: Prepare the MongoDB database**

1. Open a command prompt, and run the following command:
2. E:\MongoDB\bin\mongo.exe
3. At the MongoDB shell, run the following commands:
4. use SalesData;
5. db.createUser(
6. {
7. user: "salesadmin",
8. pwd: "Pa55w.rd",
9. roles: [ { role: "readWrite", db: "SalesData" } ]
10. }
11. );
12. use admin;
13. db.shutdownServer();
14. exit;
15. Restart the MongoDB by using net start mongodb.
16. Back in the command window, connect to the database with the new salesadmin credentials using the following command:
17. mongo.exe -u "salesadmin" -p "Pa55w.rd" --authenticationDatabase "SalesData"
18. At the MongoDB shell, run the following commands:
19. use SalesData;
20. db.testcollection.insert({testdata: 1});
21. show collections;
22. db.testcollection.find();
23. exit;
24. In the command window run the mongorestore command to import data for the task:
25. mongorestore -u "salesadmin" -p "Pa55w.rd" --db SalesData --collection customers "E:\Labfiles\Lab01\Setup\customers.bson"
26. Note that 407 customer documents have been imported.

**Task 2: Run the DocumentDB Data Migration Tool**

* Start the GUI version of the Data Migration tool **E:\dmt\bin\dtui\dtui.exe**.

**Task 3: Import data from the MongoDB database to you Cosmos DB database**

1. Use the Azure portal, navigate to the SalesData Cosmos DB database you created in exercise 1.
2. On the **Keys** blade, copy the **PRIMARY CONNECTION STRING** to the clipboard.
3. In the Data Migration tool, set the source connection string to:
4. mongodb://salesadmin:Pa55w.rd@localhost/SalesData
5. Set the Collection to **customers**.
6. For the target, set the target to **DocumentDB - Bulk import (single partition collections)**, set the connecion string to the **PRIMARY CONNECTION STRING** that you copied to the clipboard, and append **database=SalesData** to the end of the connection string.
7. Set the Collection to **customers**.
8. Set the Partition Key to **/customerid**.
9. Set the ID field to **\_id**. If you don't specify an ID, Cosmos DB has to generate a unique reference for each record. Specifying the ID therefore reduces the time to import the data.
10. Return to the Azure portal, use the Data Explorer to return the total number of documents in the customers collection.

**Question**: How many documents are returned?

**Result**: At the end of this exercise, you will have:

* Installed MongoDB and imported an existing MongoDB collection of documents.
* Installed the Data Migration tool.
* Migrated MongoDB data into an Azure Cosmos DB.
* Use the Data Explorer to view data in a documents database.

**Exercise 3: Access data by using the SQL API**

**Scenario**

To prove that the new Cosmos DB will be usable for your company, you have been asked to finish a simple menu-driven application that an intern started but didn't finish. It is designed to enable your sales team to query customer data.

The sales team have requested to be able to search for customers by either surname or the document id for the customer.

The main tasks for this exercise are as follows:

1. Examine the existing .NET application.
2. Configure the application to connect to Cosmos DB using the SQL API.
3. Finish the code for the application.
4. Test the application.

**Task 1: Examine the existing .NET application**

1. Using Visual Studio 2017, open the following solution:
2. E:\Labfiles\Lab01\Starter\QueryCustomers\QueryCustomers.sln
3. Examine the CustomerDetails.cs code. Note that there are two classes AddressInfo and CustomerInfo that represent the documents in the database.
4. Examine the Program.cs code. Note there is a worker asynchronous DoWork method responsible for performing the queries and displaying the results.

**Task 2: Configure the application to connect to Cosmos DB using the SQL API**

1. Open your Cosmos DB account in Cloud Explorer, and make a note of the Document Endpoint and Primary Master Key values.
2. Edit the app.config file, and set the EndpointUrl and PrimaryKey app settings to the Document Endpoint and Primary Master Key values.
3. Using the NuGet Package Manager, add the Microsoft.Azure.DcumentDB.Core package to the solution. This package contains the libraries that implement the SQL API. Allow the NuGet Package Manager to install all the dependencies that it requires.

**Task 3: Finish the application**

1. In the Worker.DoWork method create a DocumentClient object connecting with a Uri object.
2. In the FindCustomerByDocumentID method, use the CreateDocumentUri to build a unique Uri to a document with the id stored in the **id** variable.
3. Add a call to the ReadDocumentAsync using the created Uri and specify the customer ID held in the **custid** variable as the partition key.
4. Write the returned document to the console.
5. Add the following code to return a document by last name to the FindCustomerByLastName method.
6. string queryString = $"SELECT \* FROM {this.collection} c WHERE c.lastname = @lastname";
7. SqlParameterCollection parameters = new SqlParameterCollection()
8. {
9. new SqlParameter("@lastname", lastName)
10. };
11. SqlQuerySpec querySpec = new SqlQuerySpec()
12. {
13. QueryText = queryString,
14. Parameters = parameters

};

1. You will be creating a Uri for a collection instead of document, and will need to loop through the returned results using this code.
2. Uri collectionUri = UriFactory.CreateDocumentCollectionUri(this.database, this.collection);
3. var query = this.client.CreateDocumentQuery<CustomerInfo>(collectionUri, querySpec, new FeedOptions
4. {
5. EnableCrossPartitionQuery = true

});

1. Loop through the results in query and write the documents to the console.

**Task 4: Test the application**

1. Build the application and fix any compiler errors.
2. Run the application.
3. Test returning a document by specifying the document id and customer id for an existing customer.

**Hint**: Use the Data Explorer to find an existing document.

**Question:** What happens if you use an id that doesn't exist?

1. Test returning documents by surname.
2. Close the application when finished.

**Result**: At the end of this exercise, you will have:

* Examined the C# code for an existing .NET application.
* Configured an application to connect to Cosmos DB using the SQL API.
* Finished the code for the application.
* Tested the application.

**Exercise 4: Protect data in a Cosmos DB database**

**Scenario**

Know that you have tested your application you are ready to release it to the sales team. Before you do you have been asked to restrict certain users of the application access so that they can only read data. You need to remove the master key from the client application and create tokens to grant users specific access to the data.

The main tasks for this exercise are as follows:

1. Set permissions for users.
2. Configure and run the user creation application.
3. Create a middle tier application to authenticate users and generate resource tokens.
4. Build a client application to test this middle tier web API.
5. Close down the resources to save azure costs.

**Task 1: Set permissions for users**

1. Using Visual Studio open the **E:\Labfiles\Lab01\Starter\EnrollClient\EnrollClient.sln** solution.
2. Examine the **Main** method in the **Program.cs** file. This method prompts the user to provide a username, password, and permission either read only or all access.
3. The **CreateUser** method inside the **UserEnroller** class creates the user on the **SalesData** database and stores the username and password in **SecurityData** database.
4. The **VerifyUser** method displays the permissions granted to a specified user.

**Important**: This is a rather simplistic approach to authentication. In a production environment passwords should be hashed and encrypted.

1. After the TODO: Connect to the Cosmos DB account add the following code:

DocumentClient client = new DocumentClient(new Uri(endpointUrl), primaryKey);

1. After the comment TODO: Check whether the user already exists, add the following code. This code queries the users list for the database to determine whether a user that matches the userName parameter passed to the CreateUser method already exists:
2. Uri usersUri = new Uri($"dbs/{database}/users", UriKind.Relative);
3. string query = "SELECT \* FROM u WHERE u.id = @id";
4. SqlParameter param = new SqlParameter("@id", userName);
5. SqlQuerySpec querySpec = new SqlQuerySpec(query, new SqlParameterCollection() { param });
6. var existingUserQuery = client.CreateUserQuery(usersUri, querySpec);

var result = existingUserQuery.ToList();

1. After the comment TODO: If the user doesn't exist, then create it, add the following statements. This code creates a user with the specified user name.
2. User databaseUser = new User
3. {
4. Id = userName

};

1. After the comment TODO: Record the user's name and password in the security database (used for authenticating the user when they log in), add the following statements. This code adds a document containing the user name and password to the **SecurityData** database:
2. Uri securityCollectionUri = UriFactory.CreateDocumentCollectionUri(securityDatabase, securityCollection);
3. UserData userData = new UserData
4. {
5. UserName = userName,
6. Password = password

};

1. After the comment TODO: Set the user's permission on the specified database and collection, add the following code. These statements create a new **Permission** object specifying the privilege assigned to the user over the collection. The call to the **VerifyUser** method should display the permission that has just been assigned:
2. Permission collectionPermission = new Permission
3. {
4. PermissionMode = permission == 'A' ? PermissionMode.All : PermissionMode.Read,
5. ResourceLink = UriFactory.CreateDocumentCollectionUri(database, collection).ToString(),
6. Id = $"{userName}-{permission}"

};

**Task 2: Configure and run the application**

1. Using the Azure portal, go to the **Data Explorer** of the Cosmos DB you created in exercise 1.
2. Create a new database named **SecurityData** and add a collection named **Passwords**, with the partition key set to **/UserName** and the throughput set to 400.
3. If you haven't already saved the **URI** and **PRIMARY KEYS**, copy them from the **Keys** section in the left pane.
4. Add the copied data to the **App.config**, then build and run the application.
5. Create a user and password with all access.
6. Create a user and password with read only access.
7. Return to the **Data Explorer** and review the documents you have just created in the **SecurityData** database.

**Task 3: Authenticate users and generate resource tokens**

Note: The middle tier is a .NET web application that provides an API to return resource tokens to authenticated users. These tokens will grant access to the **Customers** collection in the **SalesData**, with the permissions created for the user.

1. Using Visual Studio open the **E:\Labfiles\Lab01\Starter\AuthenticateUsers\AuthenticateUsers.sln** solution.
2. The solution contains two projects:
   * **AuthenticationWebApi**, which implements authentication as described above.
   * **AuthenticatedCustomersClient**, which is a sample client app that uses resource tokens to connect to the Cosmos DB account.
3. In the **AuthenticateController** constructor, after the comment TODO: Retrieve the usernames and passwords from the security database, add the following code. These statements create a query that fetches every document (containing the username and password) from the **Passwords** collection in the **SecurityData** database.
4. string securityDatabase = ConfigurationManager.AppSettings["SecurityDatabase"];
5. string securityCollection = ConfigurationManager.AppSettings["SecurityCollection"];
6. Uri securityCollectionUri = UriFactory.CreateDocumentCollectionUri(securityDatabase, securityCollection);
7. var query = client.CreateDocumentQuery<UserData>(securityCollectionUri, new FeedOptions
8. {
9. EnableCrossPartitionQuery =true

});

1. After the comment Populate the users Dictionary from the username/password database, add the following code. These statements retrieve all of the security documents and cache them in a Dictionary collection named **users**:
2. users = new Dictionary<string, string>(query.Count());
3. foreach (var user in query)
4. {
5. users[user.UserName] = user.Password;

}

1. In the **Authenticate** method, after the comment TODO: Get the resource token(s) from the permission feed in the database, add the following statements. This code connects to the permission feed in the database to read all of the permissions assigned to users:
2. string database = ConfigurationManager.AppSettings["Database"];
3. string collection = ConfigurationManager.AppSettings["Collection"];
4. var permissionsUri = UriFactory.CreateUserUri(database, userName);

var permissions = await client.ReadPermissionFeedAsync(permissionsUri);

1. After the comment TODO: Find the resource token for the specified user, add the following code. When the permission was created, the ID was given the value *<username>-<permission>* (for example **Fred-R**). These statements use the ID to find the permission that matches the username, and returns the resource token generated for this permission:
2. Regex expression = new Regex($"{userName}-.");
3. string resourceToken = null;
4. foreach (var permission in permissions)
5. {
6. var matching = expression.Matches(permission.Id);
7. if (matching.Count > 0)
8. {
9. resourceToken = permission.Token;
10. return Ok(resourceToken); ;
11. }

}

1. After the comment TODO: If there was no token for the specified partition, then return HTTP status code 401 (Unauthorized), add the following statement:

return Unauthorized();

1. Edit the **Web.config file**. Set the **EndpointUrl** and **PrimaryKey** appsettings to match those of your Cosmos DB account.
2. Save all the changes to files.

**Task 4: Build a client application to test the web API**

Note: The AuthenticatedCustomersClient project uses the web API to authenticate a user, if successful it reads the resource token returned and connects to the Cosmos DB **Customers** collection.

1. In the **Program.cs** file, in the **DoWork** method, after the TODO: Use the Web API to authenticate the user and obtain a collection of security tokens comment add the following code. These statements construct an HTTP Get request Authenticate method of the web API and send it to the web application. Note that you should replace the URI in the base address with the URL of your web application:
2. HttpClient httpClient = new HttpClient();
3. httpClient.BaseAddress = new Uri("http://localhost:50281/");
4. httpClient.DefaultRequestHeaders.Accept.Clear();
5. httpClient.DefaultRequestHeaders.Accept.Add(
6. new MediaTypeWithQualityHeaderValue("application/json"));

HttpResponseMessage response = await httpClient.GetAsync($"api/Authenticate?userName={userName}&password={password}");

1. After the comment TODO: If the user was successfully authenticated, retrieve the resource token from the response, add the following statements. This code extracts the resource token from the body of the response.
2. Stream tokenStream = await response.Content.ReadAsStreamAsync();
3. StreamReader reader = new StreamReader(tokenStream);
4. var token = await Task.Factory.StartNew(() =>

JsonConvert.DeserializeObject<string>(reader.ReadToEnd()));

1. After the comment TODO: Connect to the Cosmos DB account using the resource token, add the following statements. Note that you use the token retrieved in the previous step to connect to the Cosmos DB account, and not the master key:
2. string endpointUrl = ConfigurationManager.AppSettings["EndpointUrl"];

DocumentClient client = new DocumentClient(new Uri(endpointUrl), token);

1. After the comment TODO: Read data from the collection. Add the following statements. This code uses the techniques shown in previous exercises to fetch the customers data from the **Customers** collection:
2. string database = ConfigurationManager.AppSettings["Database"];
3. string collection = ConfigurationManager.AppSettings["Collection"];
4. Uri collectionUri = UriFactory.CreateDocumentCollectionUri(database, collection);
5. var query = client.CreateDocumentQuery<CustomerInfo>(collectionUri);
6. foreach (var doc in query)
7. {
8. Console.WriteLine();
9. Console.WriteLine($"{doc.ToString()}");

}

1. After the comment TODO: Attempt to write to the collection, add the following statements. This code creates a new **CustomerInfo** object and writes it to the **Customers** collection:
2. CustomerInfo user = new CustomerInfo
3. {
4. CustomerID = new Random().Next(),
5. FirstName = "Brand",
6. MiddleName = "New",
7. LastName = "Customer"
8. };

await client.CreateDocumentAsync(collectionUri, user);

1. Edit the **App.config** file, and provide the URL of your Cosmos DB account in the **EndpointUrl** app setting.
2. Set the solution to start both projects.
3. Build and run the solution. A web browser window will appear, and a console window.
4. In the console window, enter your username and password when prompted. If you completed the previous tasks correctly, your account should have all access to the **Customers** collection. You should see a list of customers, and the new customer will be added successfully.
5. In the console enter the user with the read only access, the list of customers should be returned, then an error message stating that the user has insufficient access rights to add a new document to the collection.

**Task 5: Lab close down**

* To reduce your costs, delete the resource group containing your Cosmos DB databases.

**Result**: At the end of this exercise, you will have:

* Set permissions for users.
* Configured, run, and created users with permissions.
* Created a middle tier application to authenticate users and generate resource tokens.
* Finished a client application that uses a web API to retrieve resource tokens.
* Closed down the resources to save on Azure costs.

**Exercise 1: Create an Azure Cosmos DB database with a SQL API**

**Task 1: Prepare the environment**

1. Ensure that the **MT17B-WS2016-NAT** and **20777A-LON-DEV** virtual machines are running, and then log on to **20777A-LON-DEV** as **Administrator** with the password **Pa55w.rd**.
2. In File Explorer, go to **E:\Labfiles\Lab01**, right-click **Setup.cmd**, and then click **Run as administrator**.
3. If the **Security warning** message appears, type **R**, and then press Enter.
4. Wait until the script has completed, and then press Enter.

**Task 2: Create a document database using the Cosmos DB SQL API**

1. On the toolbar, click **Internet Explorer**.
2. In the address bar, type [**http://portal.azure.com**](http://portal.azure.com/), and then press Enter.
3. Login with your Azure pass credentials.
4. In the left pane, click **Azure Cosmos DB**.
5. On the **Azure Cosmos DB** blade, click **+ Add**.
6. On the **Create Azure Cosmos DB Account** blade, under the **Resource Group** box, click **Create new**, type **20777\_Mod01**, and then click **OK**.
7. In the **Account Name** box, type **20777a-sql-<*your name>-<the day*>**, for example **20777a-sql-sam-12**.
8. In the **API** drop-down list, click **Core (SQL)**.
9. In the **Location** drop-down list, click the region closest to your current location, click **Review + create**, and then click **Create**.
10. Wait for the Azure Cosmos DB to be created—this might take a few minutes.
11. Click **Go to resource**.
12. On the **20777a-sql-<*your name>-<the day*>** blade, click **Data Explorer**, and then click **New Database**.
13. In the **New Database** pane, in the **Database id** box, type **SalesData**, and then click **OK**.
14. On the **Data Explorer** blade, click **New Collection**.
15. In the **Add Collection** pane, under **Database id**, click **Use existing**.
16. In the **Choose an existing database** drop-down list, click **SalesData**.
17. In the **Collection Id** box, type **Customers**.
18. In the **Partition key** box, type **/customerid**.
19. In the **Throughput** box, change **400** to **5000**.
20. Under **Unique keys**, click **+ Add unique key**.
21. In the text box, type **/customerid**, and then click **OK**.

**Task 3: Add customer documents to a collection**

1. In the **SQL API** pane, under **SalesData**, expand **Customers**, and then click **Documents**.
2. On the **Documents** tab, click **New Document**.
3. Replace the default document text:
4. {
5. "id": "replace\_with\_new\_document\_id"

}

with this new document:

{

"id": "1",

"customerid": 101,

"firstname": "Orlando",

"lastname": "Gee",

"email": "orlando.gee@adventure-works.com",

"addresses": [

{ "addressline1" : "660 Lindbergh", "city" : "Saint Louis", "stateprovince" : "Missouri", "countryregion" : "United States", "postalcode" : "63103" },

{ "addressline1" : "Ontario Mills", "city" : "Ontario", "stateprovince" : "California", "countryregion" : "United States", "postalcode" : "91764" }

]

}

1. On the **Documents** tab, click **Save**.
2. Repeat steps 2 to 4 and add these documents:
3. {
4. "id": "2",
5. "customerid" : 122,
6. "title" : "Ms.",
7. "firstname" : "Caroline",
8. "middlename" : "A.",
9. "lastname" : "Vicknair",
10. "companyname" : "World of Bikes",
11. "salesperson" : "adventure-works\\jillian0",
12. "addresses": [
13. { "addressline1" : "72540 Blanco Rd.", "addressline2" : "South Side", "city" : "San Antonio", "stateprovince" : "Texas", "countryregion" : "United States", "postalcode" : "78204" }
14. ]

}

and:

{

"id": "3",

"customerid": 108,

"title" : "Mr.",

"firstname" : "Robert",

"middlename" : "R.",

"lastname" : "Vessa",

"suffix" : "Esq.",

"companyname" : "Totes & Baskets Company",

"salesperson" : "adventure-works\\jillian0",

"emailaddress" : "robert13@adventure-works.com",

"phone" : "560-555-0171"

}

**Question**: Try and add another document with the **customerid** field set to **108**, and **id** field set to **4**. What happens?

**Answer**: The Azure portal displays the message: Unique index constraint violation.

**Task 4: Perform SQL queries**

1. On the **Data Explorer** blade, click **New SQL Query**.
2. On the **Query 1** tab, note the SQL query that has been pre-populated, and then click **Execute Query**:

SELECT \* FROM c

1. Replace the above query with the following code, and then click **Execute Query**:

SELECT c.firstname, c.lastname, c.email FROM c

**Question**: As a document database is schema-less, what is returned in the customers JSON documents that do not have an **email** property?

**Answer**: Customers without the specified property do not return any data. So in the above example only the first customer has returned an email address. The next two customers only return firstname and lastname.

1. Replace the above query with the following code, and then click **Execute Query**:

SELECT c.Customerid FROM c

**Question**: Why does this query return empty data? (**Hint:** field names are case sensitive)

**Answer**: The query returns three empty documents, because none of the documents have a Customerid attribute. They all have a customerid attribute though. Azure Cosmos DB cannot perform syntax checking on what you might think of as columns, as a document database is schema-less.

1. Replace the above query with the following code, and then click **Execute Query**:

SELECT c.addresses FROM c

**Question**: Explain the format of the documents returned by this query.

**Answer**: The first customer has two addresses, the second customer has one address, and the third customer has no addresses (displayed as an empty document {})

1. Replace the above query with the following code, and then click **Execute Query**:

SELECT c.customerid, c.city FROM c

**Question**: What might you have been expecting this query to return, and why didn't it work?

**Answer**: You probably expected it to return the customerid and city for each customer, but it only displayed the customerid. This is because city is not a field in the main part of the documents - it is in the addresses sub-documents array?

**Question**: How would you display the customerid and city for each customer? Is this correct?

SELECT c.customerid, c.addresses.city FROM c

**Answer**: No, the correct answer is the following code, why?

SELECT c.customerid, c.addresses[0].city FROM c

**Note**: that addresses is an array, so you must use an array index to access sub-documents in the array (in this case, element 0 is the first sub-document). If you want to find data from all sub-documents in an array, you can use the JOIN operator, covered in more detail in module 2. If you are interested, rewrite the query as below, and execute it:

SELECT c.customerid, a.city

FROM c

JOIN a IN c.addresses

**Task 5: Configure document lifetime**

1. In the **SQL API** pane, under **Customers**, click **Scale & Settings**.
2. On the **Scale & Settings** tab, under **Time to Live**, click **On**.
3. In the **second(s)** box, type **10**, and then click **Save**.
4. Close the **Scale & Settings** tab, and wait for 10 seconds.
5. On the **Documents** tab, under the **Edit Filter** button, click the **Refresh** button.

Note: all the previous documents have been deleted.

1. In the **SQL API** pane, under **Customers**, click **Scale & Settings**.
2. On the **Scale & Settings** tab, under **Time to Live**, click **Off**, and then click **Save**.

**Task 6: Replicate the Cosmos DB Account across regions**

1. On the **20777a-sql-<*your name>-<the day*>** blade, under **Settings**, click **Replicate data globally**.
2. On the **Replicate data globally** blade, on the map, click on a hexagon containing a **+** near your current location.
3. On the map, click on a hexagon containing a **+** in another continent from your current location, and then click **Save**.

Note: that the added regions are shown to the right of the page as **READ REGIONS**. It may take a few minutes (up to a maximum of 30 minutes), to spin up a new region. The Azure portal will notify you when the regions become available.

1. When the regions have been added, click **Automatic Failover**.
2. On the **Automatic Failover** blade, click **ON**.
3. Position the cursor over one of the added regions, click and hold on the three dot drag handle, move the region to a different position and note the change in priority, and then click **OK**.
4. On the **Replicate data globally** blade, click **Save**.
5. On the **Replicate data globally** blade, click **Manual Failover**.
6. On the **Manual Failover** blade, click a read region.
7. Select the **I understand and agree to trigger a failover on my current Write Region** check box, and then click **OK**.
8. Leave Internet Explorer open for future tasks.

**Result**: At the end of this exercise, you will have:

* Created an Azure Cosmos DB database with a SQL API.
* Added documents to a document database.
* Performed a number of queries against the database.
* Configured documents Time to Live value.
* Replicated your databases data globally.

**Exercise 2: Migrate data from an existing Mongo DB database to Cosmos DB**

**Task 1: Prepare the MongoDB database**

1. On the Start menu, type **cmd**, and then press Enter.
2. At the command prompt, type the following, and then press Enter:
3. E:
4. cd MongoDB\bin
5. mongo.exe
6. At the MongoDB shell prompt **>**, type the following commands:
7. use SalesData;
8. db.createUser(
9. {
10. user: "salesadmin",
11. pwd: "Pa55w.rd",
12. roles: [ { role: "readWrite", db: "SalesData" } ]
13. }
14. );
15. use admin;
16. db.shutdownServer();
17. exit;
18. At the command prompt, type the following command to restart the MongoDB, and then press Enter:
19. net start mongodb
20. At the command prompt, type the following command, and then press Enter:
21. mongo.exe -u "salesadmin" -p "Pa55w.rd" --authenticationDatabase "SalesData"
22. At the MongoDB shell prompt **>**, type the following commands, and then press Enter:
23. use SalesData;
24. db.testcollection.insert({testdata: 1});
25. show collections;
26. db.testcollection.find();
27. exit;
28. Note that the test data has been found.
29. At the command prompt, type the following command, and then press Enter:
30. mongorestore -u "salesadmin" -p "Pa55w.rd" --db SalesData --collection customers "E:\Labfiles\Lab01\Setup\customers.bson"
31. You should see diagnostics that include the following line:
32. finished restoring SalesData.customers (407 documents)

**Task 2: Run the DocumentDB Data Migration Tool**

1. In File Explorer, go to **E:\dmt\bin\dtui**, and then double-click **dtui.exe**.
2. Leave the application running for the next task.

**Task 3: Import data from the MongoDB database to you Cosmos DB database**

1. On the **20777a-sql-<*your name>-<the day*>** blade, under **Settings**, click **Keys**.
2. Make a note of the **PRIMARY CONNECTION STRING** value.
3. In the **DocumentDB Data Migration Tool** window, on the **Welcome** page, click **Next**.
4. On the **Source** **Information** page, in the **Import from** drop-down list, click **MongoDB**.
5. In the **Connection String** box, type the following, and then click **Verify**:
6. mongodb://salesadmin:Pa55w.rd@localhost/SalesData
7. In the **Verify Connection** dialog box, click **OK**.
8. In the **Collection** box, type **customers**, and then click **Next**.
9. On the **Target** **Information** page, in the **Connection String** box, enter the value you noted earlier for **PRIMARY CONNECTION STRING** for your Cosmos DB account.
10. At the end of the string, type **Database=SalesData**, and then click **Verify**.
11. In the **Verify Connection** dialog box, click **OK**.
12. In the **Collection** box, type **Customers**.
13. IN the **Partition Key** box, type **/customerid**.
14. In the **Id Field** box, type **\_id**, and then click **Next**.
15. On the **Advanced** page, click **Next**.
16. On the **Summary** page, click **Import**.

**Question**: How many documents are transfered?

**Answer**: 407 documents

1. Close the DocumentDB Data Migration Tool window.

**Result**: At the end of this exercise, you will have:

* Installed MongoDB and imported an existing MongoDB collection ofdocuments.
* Installed the DocumentDB Data Migration Tool.
* Migrated MongoDB data into an Azure Cosmos DB.
* Use the Data Explorer to view data in a documents database.

**Exercise 3: Access data by using the SQL API**

**Task 1: Examine the existing .NET application**

1. On the Start menu, click **Visual Studio 2017**.
2. In Visual Studio 2017, on the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, go to **E:\Labfiles\Lab01\Starter\QueryCustomers**, click **QueryCustomers.sln**, and then click **Open**.
4. In Solution Explorer, double-click **CustomerDetails.cs**.
5. Note that there are two classes **AddressInfo** and **CustomerInfo** that represent the documents in the database.
6. In Solution Explorer, double-click **Program.cs**.
7. Note the **Main** method uses the asynchronous **DoWork** method to perform the queries and display the results.

**Task 2: Configure the application to connect to Cosmos DB using the SQL API**

1. Return to the Azure portal in Internet Explorer.
2. On the **20777a-sql-<*your name>-<the day*>** blade, under **Settings**, click **Keys**.
3. Make a note of the **URI**, and **PRIMARY KEY** values.
4. Return to Visual Studio.
5. In Solution Explorer, double-click **App.config**.
6. In App.config, in the **value** attribute of the **EndpointUrl** key, paste the **URI** value you noted earlier, replacing the text **enter URI here**.
7. In the **value** attribute of the **PrimaryKey** key, paste the **PRIMARY KEY** value you noted earlier, replacing the text **enter Primary Key here**.

**Task 3: Finish the application**

1. In Solution Explorer, double-click **Program.cs**.
2. Under the **// TODO: Connect to the Cosmos DB database** comment, type the following code:

this.client = new DocumentClient(new Uri(endpointUrl), primaryKey);

1. In the **FindCustomerDocumentByID** method, under the **// TODO: Construct the URI for this document** comment, type the following code:

Uri docUri = UriFactory.CreateDocumentUri(this.database, this.collection, id);

1. Under the **// TODO: Fetch the document** comment, type the following code:
2. var documentResponse = await client.ReadDocumentAsync<CustomerInfo>(docUri, new RequestOptions()
3. {
4. PartitionKey = new PartitionKey(custid)

});

1. Under the **// TODO: Display the contents of the document** comment, type the following code:
2. Console.WriteLine(documentResponse.Document);

Console.WriteLine();

1. In the **FindCustomerByLastName** method, under the **// TODO: Construct a query to find matching customers**comment, type the following code:
2. string queryString = $"SELECT \* FROM {this.collection} c WHERE c.lastname = @lastname";
3. SqlParameterCollection parameters = new SqlParameterCollection()
4. {
5. new SqlParameter("@lastname", lastName)
6. };
7. SqlQuerySpec querySpec = new SqlQuerySpec()
8. {
9. QueryText = queryString,
10. Parameters = parameters

};

1. Under the **// TODO: Specify the collection to run the query against** comment, type the following code:
2. Uri collectionUri = UriFactory.CreateDocumentCollectionUri(this.database, this.collection);
3. var query = this.client.CreateDocumentQuery<CustomerInfo>(collectionUri, querySpec, new FeedOptions
4. {
5. EnableCrossPartitionQuery = true

});

1. Under the **// TODO: Run the query, and display the list of matching customers** comment, type the following code:
2. foreach (var doc in query)
3. {
4. Console.WriteLine($"{doc.ToString()}");

}

**Task 4: Test the application**

1. In the Azure portal in Internet Explorer, on the **20777a-sql-<*your name>-<the day*>** blade, click **Data Explorer**.
2. In the **SQL API** pane, expand **SalesData**, expand **Customers**, click **Documents**, and then click a document.
3. In the document, note the value for the **id** and the **customerid** fields.
4. Return to Visual Studio.
5. Press F5 to build and run the application.
6. Type **a**, type the document id noted in step 3, and then press Enter.
7. Type the customer id noted in step 3, and then press Enter.
8. Examine the document that is returned.
9. Type **a**, for the document id, type some random numbers , and then press Enter.
10. For the customer id, type some random numbers, and then press Enter.

**Question:** What happens if you use an id or customer id that doesn't exist?

**Answer:** An error is returned:

{"Errors":["Resource Not Found"]}

1. Type **b**, type **Gee**, and then press Enter.
2. Examine the document that is returned.
3. Type **b**, type a random surname, and then press Enter.

**Question:** What happens if you use a surname that doesn't exist?

**Answer:** Nothing is returned. This isn't an error condition as the code can handle zero to many matches on surname. However, by using the UriFactory.CreateDocumentUri method it expects a valid Id. This errors when the application tries to generate the URI.

1. Type **x** to close the application.
2. On the **File** menu, click **Close Solution**.
3. Leave Visual Studio open for the next exercise.

**Result**: At the end of this exercise, you will have:

* Examined the C# code for an existing .NET application.
* Configured an application to connect to Cosmos DB using the SQL API.
* Finished the code for the application.
* Tested the application.

**Exercise 4: Protect data in a Cosmos DB database**

**Task 1: Set permissions for users**

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, go to **E:\Labfiles\Lab01\Starter\EnrollClient**, click **EnrollClient.sln**, and then click **Open**.
3. In Solution Explorer, double-click **Program.cs**.
4. In Program.cs, examine the **Main** method. This method prompts the user to provide a username, password, and permission either read only or all access.
5. In Solution Explorer, double-click **UserEnroller.cs**.
6. In UserEnroller.cs, the **CreateUser** method inside the **UserEnroller** class creates the user on the **SalesData** database and stores the username and password in **SecurityData** database. The **VerifyUser** method displays the permissions granted to a specified user.
7. In the **CreateUser** method, under the comment **// TODO: Connect to the Cosmos DB account**, type the followin code:

DocumentClient client = new DocumentClient(new Uri(endpointUrl), primaryKey);

1. Under the comment **// TODO: Check whether the user already exists**, type the following code:
2. Uri usersUri = new Uri($"dbs/{database}/users", UriKind.Relative);
3. string query = "SELECT \* FROM u WHERE u.id = @id";
4. SqlParameter param = new SqlParameter("@id", userName);
5. SqlQuerySpec querySpec = new SqlQuerySpec(query, new SqlParameterCollection() { param });
6. var existingUserQuery = client.CreateUserQuery(usersUri, querySpec);

var result = existingUserQuery.ToList();

1. Under the comment **// TODO: If the user doesn't exist, then create it**, type the following code:
2. User databaseUser = new User
3. {
4. Id = userName

};

1. Under the comment **// TODO: Record the user's name and password in the security database (used for authenticating the user when they log in)**, type the following code:
2. Uri securityCollectionUri = UriFactory.CreateDocumentCollectionUri(securityDatabase, securityCollection);
3. UserData userData = new UserData
4. {
5. UserName = userName,
6. Password = password

};

1. Under the comment **// TODO: Set the user's permission on the specified database and collection**, type the following code:
2. Permission collectionPermission = new Permission
3. {
4. PermissionMode = permission == 'A' ? PermissionMode.All : PermissionMode.Read,
5. ResourceLink = UriFactory.CreateDocumentCollectionUri(database, collection).ToString(),
6. Id = $"{userName}-{permission}"

};

**Task 2: Configure and run the application**

1. In Internet Explorer, on the **20777a-sql-<*your name>-<the day*>** blade , click **Data Explorer**, and then click **New Database**.
2. In the **New Database** blade, under **Database id**, type **SecurityData**, and then click **OK**.
3. Right-click **SecurityData**, and then click **New Collection**.
4. On the **Add Collection** blade, under the **Collection Id**, type **Passwords**.
5. Under the **Partition key**, type **/UserName**.
6. Under **Throughput**, type **400**, and then click **OK**.
7. In Visual Studio, in Cloud Explorer, click **Refresh All**.
8. In Solution Explorer, double-click **App.config**.
9. In App.config, in the **Value** attribute of the **EndpointUrl** key, paste the **URI** value you noted earlier, replacing the text **enter URI here**.
10. In the **Value** attribute of the **PrimaryKey** key, paste the **Primary Master Key** value you noted earlier, replacing the text **enter Primary Key here**.
11. Press F5 to build and run the solution.
12. At the command prompt, after **User name:**, type **admin**, and then press Enter.
13. At the command prompt, after **Password:**, type **adm1n**, and then press Enter.
14. At the command prompt, after **Permission R(ead)/A(ll):**, type **A**.
15. At the command prompt, after **Press q to quit, any other key to add a new user**, press Enter.
16. At the command prompt, after **User name:**, type **test**, and then press Enter.
17. At the command prompt, after **Password:**, type **t3st**, and then press Enter.
18. At the command prompt, after **Permission R(ead)/A(ll):**, type **R**.
19. At the command prompt, type **q**.
20. On the **File** menu, click **Close Solution**.
21. In Internet Explorer, on the **Data Explorer** blade, in the **SQL API** pane, expand **SecurityData**, expand **Passwords**, and then click **Documents**.
22. On the **Documents** tab, click any of the **ids** to see the documents contents.

**Task 3: Authenticate users and generate resource tokens**

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, go to **E:\Labfiles\Lab01\Starter\AuthenticateUsers**, click **AuthenticateUsers.sln**, and then click **Open**.
3. The solution contains two projects:
   * **AuthenticationWebApi**, which implements authentication as described above.
   * **AuthenticatedCustomersClient**, which is a sample client app that uses resource tokens to connect to the Cosmos DB account.
4. In Solution Explorer, expand **AuthenticationWebApi**, expand **Controllers**, and then double-click **AuthenticateController.cs**.
5. In the **AuthenticateController** constructor, under the comment **// TODO: Retrieve the usernames and passwords from the security database**, type the following code:
6. string securityDatabase = ConfigurationManager.AppSettings["SecurityDatabase"];
7. string securityCollection = ConfigurationManager.AppSettings["SecurityCollection"];
8. Uri securityCollectionUri = UriFactory.CreateDocumentCollectionUri(securityDatabase, securityCollection);
9. var query = client.CreateDocumentQuery<UserData>(securityCollectionUri, new FeedOptions
10. {
11. EnableCrossPartitionQuery =true

});

1. Under the comment **// TODO: Populate the users Dictionary from the username/password database**, type the following code:
2. users = new Dictionary<string, string>(query.Count());
3. foreach (var user in query)
4. {
5. users[user.UserName] = user.Password;

}

1. In the **Authenticate** method, under the comment **// TODO: Get the resource token(s) from the permission feed in the database**, type the following code:
2. string database = ConfigurationManager.AppSettings["Database"];
3. string collection = ConfigurationManager.AppSettings["Collection"];
4. var permissionsUri = UriFactory.CreateUserUri(database, userName);

var permissions = await client.ReadPermissionFeedAsync(permissionsUri);

1. Under the comment **// TODO: Find the resource token for the specified user**, type the following code:
2. Regex expression = new Regex($"{userName}-.");
3. string resourceToken = null;
4. foreach (var permission in permissions)
5. {
6. var matching = expression.Matches(permission.Id);
7. if (matching.Count > 0)
8. {
9. resourceToken = permission.Token;
10. return Ok(resourceToken); ;
11. }

}

1. Under the comment **// TODO: If there was no token for the specified partition, then return HTTP status code 401 (Unauthorized)**, type the following code:

return Unauthorized();

1. In Solution Explorer, double-click **Web.config**.
2. In Web.config, in the **Value** attribute of the **EndpointUrl** key, paste the **URI** value you noted earlier, replacing the text **enter URI here**.
3. In the **Value** attribute of the **PrimaryKey** key, paste the **Primary Master Key** value you noted earlier, replacing the text **enter Primary Key here**.
4. On the **File** menu, click **Save All**.

**Task 4: Build a client application to test the web API**

1. In Solution Explorer, under **AuthenticatedCustomersClient**, double-click **Program.cs**.
2. In Program.cs, in the **DoWork** method, under the comment **// TODO: Use the Web API to authenticate the user and obtain a collection of security tokens**, type add the following code:
3. HttpClient httpClient = new HttpClient();
4. httpClient.BaseAddress = new Uri("http://localhost:50281/");
5. httpClient.DefaultRequestHeaders.Accept.Clear();
6. httpClient.DefaultRequestHeaders.Accept.Add(
7. new MediaTypeWithQualityHeaderValue("application/json"));

HttpResponseMessage response = await httpClient.GetAsync($"api/Authenticate?userName={userName}&password={password}");

1. Under the comment **// TODO: If the user was successfully authenticated, retrieve the resource token from the response**, type add the following code:
2. Stream tokenStream = await response.Content.ReadAsStreamAsync();
3. StreamReader reader = new StreamReader(tokenStream);
4. var token = await Task.Factory.StartNew(() =>

JsonConvert.DeserializeObject<string>(reader.ReadToEnd()));

1. Under the comment **// TODO: Connect to the Cosmos DB account using the resource token**, type add the following code:
2. string endpointUrl = ConfigurationManager.AppSettings["EndpointUrl"];

DocumentClient client = new DocumentClient(new Uri(endpointUrl), token);

1. Under the comment **// TODO: Read data from the collection**, type add the following code:
2. string database = ConfigurationManager.AppSettings["Database"];
3. string collection = ConfigurationManager.AppSettings["Collection"];
4. Uri collectionUri = UriFactory.CreateDocumentCollectionUri(database, collection);
5. var query = client.CreateDocumentQuery<CustomerInfo>(collectionUri);
6. foreach (var doc in query)
7. {
8. Console.WriteLine();
9. Console.WriteLine($"{doc.ToString()}");

}

1. After the comment **// TODO: Attempt to write to the collection**, add the following statements. This code creates a new **CustomerInfo** object and writes it to the **Customers** collection:
2. CustomerInfo user = new CustomerInfo
3. {
4. CustomerID = new Random().Next(),
5. FirstName = "Brand",
6. MiddleName = "New",
7. LastName = "Customer"
8. };

await client.CreateDocumentAsync(collectionUri, user);

1. In Solution Explorer, under **AuthenticatedCustomersClient**, double-click **App.config**.
2. In App.config, in the **Value** attribute of the **EndpointUrl** key, paste the **URI** value you noted earlier for your Cosmos DB account, replacing the text **enter URI here**.
3. In Solution Explorer, right-click **Solution 'AuthenticatedUsers' (2 projects)**, and then click **Set Startup Projects**.
4. In the **Solution 'AuthenticatedUsers' Property Pages** dialog box, click **Multiple startup projects**.
5. In both **Action** boxes, click **Start**, and then click **OK**.
6. Press F5 to build and run the solution.
7. A web browser window will appear, and a console window.
8. In the console window, under **User name**, type **admin**, and then press Enter.
9. In the console window, under **Password**, type **adm1n**, and then press Enter.
10. Note that the customers you added earlier are listed, and a new document has been added, and then press Enter.
11. In the console window, under **User name**, type **test**, and then press Enter.
12. In the console window, under **Password**, type **t3st**, and then press Enter.
13. Note that the customers you added earlier plus the new empty one is listed. There is also an error, as the test user doesn't have the permissions to write data to the database.
14. Press Enter to close the console window.
15. Close the Home Page, and then close Visual Studio.

**Task 5: Lab close down**

1. To reduce your costs delete the resource group containing your Cosmos DB databases.
2. In Internet Explorer, in the left pane, click **Resource groups**.
3. Right-click **20777\_Mod01**, and then click **Delete resource group**.
4. On the **Are you sure you want to delete "20777\_Mod01"?** blade, in the **Type the resource group name:** box, type **20777\_Mod01**, and then click **Delete**.

**Result**: At the end of this exercise, you will have:

* Set permissions for users.
* Configured, run, and created users with permissions.
* Created a middle tier application to authenticate users and generate resource tokens.
* Finished a client application that uses a web API to retrieve resource tokens.
* Closed down the resources to save on Azure costs.

**Module 2: Designing and Implementing SQL API Database Applications**

* [Module 2: Designing and Implementing SQL API Database Applications](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#module-2-designing-and-implementing-sql-api-database-applications)
  + [Lab: Designing and Implementing a Document Database](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#lab-designing-and-implementing-a-document-database)
    - [Lab Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#lab-scenario)
    - [Objectives](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#objectives)
    - [Lab Setup](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#lab-setup)
  + [Exercise 1: Design the Document Structure for the Product Catalog](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#exercise-1-design-the-document-structure-for-the-product-catalog)
    - [Exercise 1 Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#exercise-1-scenario)
    - [Task 1: Prepare the Environment](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-1-prepare-the-environment)
    - [Task 2: Analyze the requirements](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-2-analyze-the-requirements)
    - [Task 3: Design the Documents and Partitioning Strategy](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-3-design-the-documents-and-partitioning-strategy)
    - [Task 4: Create a Document Database Using the Cosmos DB SQL API](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-4-create-a-document-database-using-the-cosmos-db-sql-api)
  + [Exercise 2: Import Data into the SalesData Database](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#exercise-2-import-data-into-the-salesdata-database)
    - [Exercise 2 Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#exercise-2-scenario)
    - [Task 1: Import Category and Subcategory Data into the Cosmos DB Database](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-1-import-category-and-subcategory-data-into-the-cosmos-db-database)
    - [Task 2: Import Product Data into the Cosmos DB Database](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-2-import-product-data-into-the-cosmos-db-database)
  + [Exercise 3: Querying Product Catalog Information](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#exercise-3-querying-product-catalog-information)
    - [Exercise 3 Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#exercise-3-scenario)
    - [Task 1: Create the Adventure-Works Web Application](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-1-create-the-adventure-works-web-application)
    - [Task 2: Define the Models and ViewModels for the Web Application](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-2-define-the-models-and-viewmodels-for-the-web-application)
    - [Task 3: Implement a Repository Class that Connects to the Cosmos DB Database](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-3-implement-a-repository-class-that-connects-to-the-cosmos-db-database)
    - [Task 4: Create the Products Controller and Initial View](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-4-create-the-products-controller-and-initial-view)
    - [Task 5: Extend the Products Controller to Fetch and Display Product Data](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-5-extend-the-products-controller-to-fetch-and-display-product-data)
  + [Exercise 4: Maintaining Stock Levels in the Product Catalog](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#exercise-4-maintaining-stock-levels-in-the-product-catalog)
    - [Exercise 4 Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#exercise-4-scenario)
    - [Task 1: Create a Collection and Repository for Storing Back Orders](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-1-create-a-collection-and-repository-for-storing-back-orders)
    - [Task 2: Extend the Repository Class to Enable Document Creation and Update](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-2-extend-the-repository-class-to-enable-document-creation-and-update)
    - [Task 3: Add Buy Functionality to the Product Details View](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-3-add-buy-functionality-to-the-product-details-view)
    - [Task 4: Lab close down](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_02.md#task-4-lab-close-down)

**Lab: Designing and Implementing a Document Database**

**Lab Scenario**

You have been asked to design an online retail system for a global organization that has thousands of product lines and handles requests from tens of thousands of customers simultaneously.

* Each customer has his or her own account. The customer account can include personal data (name, shipping addresses, preferred payment methods, security data (for authentication purposes), and other general data).
* Customers must be able to search the product catalog quickly, searching by keyword, optionally filtering by product category and sub-category, and sorting the results by factors such as price, popularity, review ratings, etc. The system should also give an indication of whether an item is currently in stock.

In this lab, you will design and implement a schema for the product catalog in Cosmos DB.

**Note**: For simplicity, this lab assumes that you will store the product catalog and product category information in two separate homogenous collections. This is not a best-practice; labs later in the course will use a single heterogenous collection.

**Objectives**

At the end of this lab, you should be able to:

* Design a document schema.
* Import data into a document collection.
* Query a document collection.
* Modify and delete data in a document collection.

**Lab Setup**

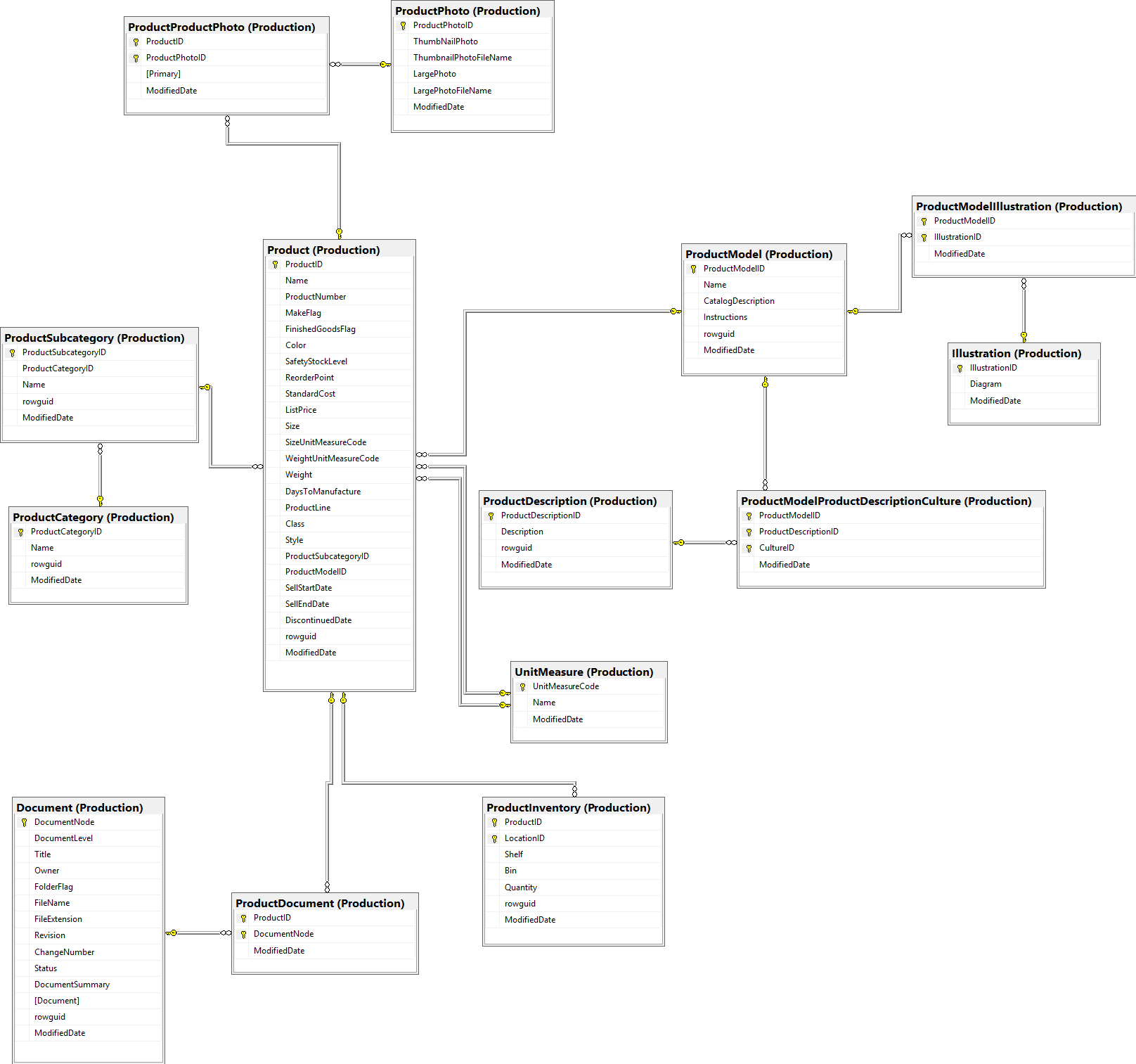
* **Estimated time**: 90 minutes
* **Virtual machine**: 20777A-LON-DEV
* **User name**: Administrator
* **Password**: Pa55w.rd

**Exercise 1: Design the Document Structure for the Product Catalog**

**Exercise 1 Scenario**

The application requires a product catalog; products must be organized by category and sub-category. You will need to consider how to handle product category and sub-category information (including current stock levels). You will also create a SQL API database and collection that matches their design.

Adventure Works Ltd. currently store product information in a SQL Server database running on premises, but are migrating to a Cosmos DB solution running in Azure. Product catalog information is held in a highly normalized in the SQL Server database, as shown in the following image:

[](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/image1.png)

The main tasks for this exercise are as follows:

1. Prepare the environment.
2. Analyze the requirements.
3. Design the documents and partitioning strategy.
4. Create a document database using the Cosmos DB SQL API.

**Task 1: Prepare the Environment**

1. Using File Explorer, in **E:\Labfiles\Lab02\Starter**, run **Setup.cmd** as **Administrator**.
2. Using the Azure portal, create a new Storage account.
   * Add the account to the **20777\_Mod02** resource group; create this resource group if it does not exist.
   * Create a new blob container. Grant anonymous read access to blobs in the container.
   * Make a note of the **Connection string** for the storage account.
3. Using the Azure portal, create a new Cosmos DB account using the SQL API.
   * Add the account to the **20777\_Mod02** resource group.
   * Create a new database named **SalesData**.
   * Make a note of the **URI** and **Primary Key** of the Cosmos DB account.

**Task 2: Analyze the requirements**

While the existing set of tables is excellent for an online transaction processing system, it is not efficient for the query intensive operations required by a retail system, where many thousands of customers distributed around the world could be browsing the catalog simultaneously.

The following query shows the SQL required to retrieve the list of product categories and subcategories:

SELECT c.Name AS category, s.Name AS subcategory

FROM Production.ProductCategory c

JOIN Production.ProductSubcategory s

ON s.ProductCategoryID = c.ProductCategoryID

The following query shows the SQL required to retrieve the data for each product:

SELECT p.ProductID, p.Name, p.ProductNumber, p.Color, p.ListPrice, p.Size, pum.Name As SizeUnit,

p.Weight, pw.Name As WeightUnit, psc.Name As SubcategoryName, pc.Name As CategoryName, i.Quantity,

d.Title, d.DocumentSummary, d.Document, d.FileName AS DocumentFileName, pm.Name AS ModelName,ill.Diagram,

ph.ThumbnailPhoto, ph.ThumbnailPhotoFileName, ph.LargePhoto, ph.LargePhotoFileName, pds.Description

FROM Production.Product p

LEFT JOIN Production.ProductSubcategory psc

ON p.ProductSubCategoryID = psc.ProductSubcategoryID

LEFT JOIN Production.ProductCategory pc

ON psc.ProductCategoryID = pc.ProductCategoryID

LEFT JOIN(SELECT ProductID, SUM(Quantity) AS Quantity FROM Production.ProductInventory GROUP BY ProductID) AS i

ON p.ProductID = i.ProductID

LEFT JOIN Production.UnitMeasure pum

ON p.SizeUnitMeasureCode = pum.UnitMeasureCode

LEFT JOIN Production.UnitMeasure pw

ON p.WeightUnitMeasureCode = pw.UnitMeasureCode

LEFT JOIN Production.ProductDocument pd

ON p.ProductID = pd.ProductID

LEFT JOIN Production.Document d

ON pd.DocumentNode = d.DocumentNode

LEFT JOIN Production.ProductModel pm

ON p.ProductModelID = pm.ProductModelID

LEFT JOIN Production.ProductModelIllustration pmi

ON pm.ProductModelID = pmi.ProductModelID

LEFT JOIN Production.Illustration ill

ON pmi.IllustrationID = ill.IllustrationID

LEFT JOIN Production.ProductProductPhoto ppp

ON P.ProductID = ppp.ProductID

LEFT JOIN Production.ProductPhoto ph

ON ppp.ProductPhotoID = ph.ProductPhotoID

LEFT JOIN Production.ProductModelProductDescriptionCulture pmdc

ON pm.ProductModelID = pmdc.ProductModelID AND pmdc.CultureID = 'en'

LEFT JOIN Production.ProductDescription pds

ON pmdc.ProductDescriptionID = pds.ProductDescriptionID

The document design for Cosmos DB should reflect a denormalized strategy that reduces search and response times for queries. Specifically, Adventure-Works requires a design that supports the following queries:

* List category/subcategory groups
* List products by category/subcategory
* Find products by model name
* Find products by name
* Find products by product number

**Task 3: Design the Documents and Partitioning Strategy**

The database administrator has provided you with a copy of the production SQL Server database with anonymized personally identifying information. Before you start to design your documents, you will open the database using the **(localdb)** SQL Server instance installed with Visual Studio.

1. In Visual Studio, use **Server Explorer** to open **AdventureWorks2016\_Data.mdf** in **E:\Labfiles\Lab02\Data**.
2. Run the SQL query given for category/subcategory listings in **Task 1** above, then design a document structure for category/subcategory groups.
3. Run the SQL query for product information given in **Task 1** above. Examine the output, then design a document structure for the information.

**Question**: What different approaches to the document design did you consider; what are their advantages and disadvantages?

**Question**: Will you store all the documents in one collection, or use more than one collection?

**Question**: The product data includes binary information in the form of Word documents and image files. How will you accommodate these files in Cosmos DB?

1. Identify a partitioning strategy for the product data.

**Question**: Assuming that the collection was going to scale to many millions of products, how would you partition the products collection?

**Question**: Will the product category/subcategory documents benefit from partitioning?

**Task 4: Create a Document Database Using the Cosmos DB SQL API**

1. Add a new collection named **ProductCatalog** to the **SalesData** database. Use unlimited storage, and **1000** RU/s throughput. Specify the partitioning key that you have selected. Specify **/productnumber** as the unique key.
2. Add a new collection named **ProductCategories** to the **SalesData** database. Use fixed storage and **1000** RU/s throughput. Specify **/subcategory** as the unique key.

**Result**: At the end of this exercise, you should have created a database and collection to hold the product catalog, and designed the document schema.

**Exercise 2: Import Data into the SalesData Database**

**Exercise 2 Scenario**

The main tasks for this exercise are as follows:

1. Import category and subcategory data into the Cosmos DB database.
2. Import product data into the Cosmos DB database.

**Task 1: Import Category and Subcategory Data into the Cosmos DB Database**

1. Query the **AdventureWorks** database to generate a JSON file of the category/subcategory data. Save the file as **E:\Labfiles\Lab02\categories.json**. Use this query as a basis for your file:
2. SELECT c.Name AS category, s.Name AS subcategory
3. FROM Production.ProductCategory c
4. JOIN Production.ProductSubcategory s
5. ON s.ProductCategoryID = c.ProductCategoryID

FOR json path

1. Using the Azure Portal Data Explorer, upload the **categories.json** file that you created in the last step to the **ProductCategories** collection.
2. Verify that 38 rows were added to the collection.

**Task 2: Import Product Data into the Cosmos DB Database**

The product data is more complex than the category/subcategory information; the target document structure contains subdocuments, and some information—such as images and other illustrations—must be uploaded to blob storage and linked from the product documents. For these reasons, the developers at Adventure Works have decided to create a custom application to retrieve the data from SQL Server, format it, and upload it to the ProductCatalog collection using the SQL API.

1. Open the the **MigrateProductData** solution in **E:\Labfiles\Lab02\Starter\MigrateProductData**.
2. Edit the **App.config** file. Set the **EndpointUrl** and **PrimaryKey** settings to the values you recorded for the Cosmos DB account in **Exercise 1** above. Set the **BlobStorageConnectionString** and **BlobContainer** settings to the values for your storage account in **Exercise 1** above.
3. Open the **Program.cs** file. This file contains the skeleton of the application. When it is complete, the **DoLoad** method will retrieve the data from SQL Server, create and format a new document for each row, upload the binary data in the row into blob storage, and then add the row (now containing references to the binary data in the form of blob storage URLs) to the **ProductCatalog** collection.
   * The helper method, **GetData** runs the SQL Server query shown earlier in this lab and returns a **SqlDataReader**object for fetching the data a row at a time.
   * You will use the **UploadDataToBlobStorageAsync** method to store the binary data for an item in blob storage.
   * The **UploadDocumentToCosmosD** method will save the document to the collection in Cosmos DB.
4. Edit the **Product.cs** file. You will define the structure of the documents and subdocuments required for the Cosmos DB database in this file.
   * At the comment **TODO: Specify fields for the Product document** add a class definition for your **Product**document that you designed in **Exercise 1**.
   * At the comment **TODO: Specify fields for the ProductCategoryData subdocument** add a class definition for your **ProductCategoryData** subdocument.
   * At the comment **TODO: Specify fields for the ProductDocumentData subdocument** add a class definition for your **ProductDocumentData** subdocument.
   * At the comment **TODO: Specify fields for the ProductImageData subdocument** add a class definition for your **ProductImageData** subdocument.
5. Edit the **Program.cs** file:
   * At the comment **TODO: Create a client to connect to the Cosmos DB database** add a statement to create a new instance of the **DocumentClient** class called **client**. Use the **endpointURL** and **primaryKey** string variables to configure the instance.
   * At the comment **TODO: Connect to Azure Blob Storage** add the following code to create a connection to your blob container using the Windows Azure Storage API:
   * if (!CloudStorageAccount.TryParse(this.blobStorageConnectionString, out CloudStorageAccount storageAccount))
   * {
   * Console.WriteLine("Invalid storage connection string");
   * return;
   * }

this.blobClient = storageAccount.CreateCloudBlobClient();

* + At the comment **TODO: Create a document from the current row retrieved from SQL Server** add code to create a new instance of the **Product** class called **productDocument** populated from the properties of a row in the result set returned by the SQL Server product query, held in the **reader** object. If necessary, use the **UploadDataToBlobStorageAsync** method to upload data to blob storage.
  + At the comment **TODO: Upload the document to the collection in Cosmos DB** add the following code to upload the document object to Cosmos DB:
  + if (await UploadDocumentToCosmosDB(productDocument))
  + {
  + Console.WriteLine($"Added document {productDocument.ProductID} for {productDocument.ProductName}: {productDocument.Description}");
  + }
  + else
  + {
  + Console.WriteLine($"Upload failed for {productDocument.ProductID}: {productDocument.ProductName}");

}

**Note**: This block calls the **UploadDocumentToCosmosDB** helper method to save the document to Cosmos DB. The **UploadDocumentToCosmosDB** method returns a boolean value indicating whether the document was saved successfully or not.

* + At the comment **TODO: If the data is NULL, return an empty string**. Add the following code after this comment. Note all products have images or other associated binary data, and in this case the data retrieved from SQL Server will be **DBNull**. This code simply returns an empty string for these situations, and the corresponding field in the **Product** document will be empty:
  + if (string.Compare(data.GetType().Name, "DBNull") == 0)
  + {
  + return string.Empty;

}

* + After the comment **TODO: Otherwise cast the data into a byte array**, add the following statement. This conversion is necessary because the Azure Storage API expects binary data to be passed as a stream of bytes:

var byteData = data as byte[];

* + At the comment **TODO: If the cast failed, the data probably contains a string, so use GetBytes to convert it into a byte array**, add the following statement. Some of the binary data in the SQL Server database is stored as strings. In this case, the previous cast will fail and the **byteData** array will be **null**. This code handles the situation by converting the string into a byte array:
  + if (byteData == null)
  + {
  + byteData = Encoding.ASCII.GetBytes(data.ToString());

}

* + At the comment **TODO: Get a reference to the blob container and a blob in Azure storage. The blob will hold the data being uploaded**, add the statements shown below. This code uses the Azure Storage API to obtain a reference to your blob storage container:
  + var cloudBlobContainer = this.blobClient.GetContainerReference(this.blobContainer);

var blockBlob = cloudBlobContainer.GetBlockBlobReference(name);

* + At the comment **TODO: Upload the data to the blob**, add the following statement. This statement streams the data from the byte array into the blob:

await blockBlob.UploadFromByteArrayAsync(byteData, 0, byteData.Length);

* + At the comment **TODO: Return the URI of the new blob as a string**, add the following statement which obtains the URL of the newly created blob and returns it to the caller:

return blockBlob.Uri.ToString();

* + At the comment **TODO: Add the document to the Cosmos DB database**, add the following statements. This code uses the SQL API to add the document to the collection in Cosmos DB. It examines the status code returned by the operation. If it is **Created** then the method returns **true** to indicate that the document was uploaded successfully. If the status has some other value, then the upload must have failed and the method returns **false**:
  + Uri productCollection = UriFactory.CreateDocumentCollectionUri(this.database, this.collection);
  + var response = await this.client.CreateDocumentAsync(productCollection, productDocument);

return (response.StatusCode == System.Net.HttpStatusCode.Created);

1. Build and run the application. Verify that it retrieves the data from SQL Server and uploads the documents to Cosmos DB without reporting any errors.
2. Using the Azure portal, go to Data Explorer for the **ProductCatalog** collection and examine some of the documents. Click the link for any of the images—they should be displayed by the browser.

**Result**: At the end of this exercise, you should have populated your Cosmos DB database with product catalog information.

**Exercise 3: Querying Product Catalog Information**

**Exercise 3 Scenario**

Customers will browse the product catalog by using a web application. The application will enable customers to search for products by category, subcategory, name, model, or product number—as described in the list of requirements at the start of this lab. You will build this web application in this exercise.

The main tasks for this exercise are as follows:

* Create the Adventure-Works web Application.
* Define the models and view models for the web application.
* Implement a repository class that connects to the Cosmos DB database.
* Create the products controller and initial view.
* Extend the products controller to fetch and display product data.

**Task 1: Create the Adventure-Works Web Application**

* In Visual Studio, create a new project using the **ASP.NET Web Application (.NET framework)** template. Call the project **Adventure-Works** and use the location **E:\Labfiles\Lab02**.
  1. In the **New ASP.NET Web Application** dialog box, select **MVC**.
  2. On the **Tools** menu, click **NuGet Package Manager**, and then click **Manage NuGet Packages for Solution**.
  3. On the **Browse** tab, enter **DocumentDB**, select **Microsoft.Azure.DocumentDB**, and add the package to the project.
  4. On the **Updates** tab, install any updates for out-of-date packages.

**Task 2: Define the Models and ViewModels for the Web Application**

1. Add a new C# class to the **Models** folder. Name the class file **Product.cs**.
2. Add the same class definitions that you used in the data import application to the **Product.cs** file. These are the same classes that you used to migrate the data to Cosmos DB, and they specify the structure of the documents and subdocuments in the **ProductCatalog** collection. Add definitions for **Product**, **ProductCategoryData**, **ProductDocumentData**, and **ProductImageData**.
3. Add the following **using** directive to the list at the top of the file. The **JsonProperty** attribute is defined in this namespace:

using Newtonsoft.Json;

1. Add another class file to the **Models** folder, named **ViewModels.cs**.
2. In the **ViewModels.cs** file, replace the **ViewModels** class with the following class definitions. You will use these classes to work with the MVC framework to display data and retrieve user input:
3. public class ProductCategoryViewModel
4. {
5. public string CategoryName { get; set; }
6. public IEnumerable<SelectListItem> Categories { get; set; }
7. }
8. public class ProductSubcategoryViewModel
9. {
10. public string SubcategoryName { get; set; }
11. public IEnumerable<SelectListItem> Subcategories { get; set; }
12. }
13. public class ProductViewModel
14. {
15. public string SearchString { get; set; }
16. public IEnumerable<Product> Products { get; set; }
17. public ProductCategoryViewModel SelectableCategories { get; set; }
18. public ProductSubcategoryViewModel SelectableSubcategories { get; set; }

}

1. Add the following **using** directive to the list at the top of the file:

using System.Web.Mvc;

This namespace contains the **SelectListItem** class used by the MVC framework to support drop-down lists. In this application, you will use drop-down lists to display catagory and subcategory information. The item the customer selects will be transferred to the **CategoryName** or **SubcategoryName** fields in the corresponding class by the MVC framework. The **ProductViewModel** class incorporates **ProductCategoryViewModel** and **ProductSubcategoryViewModel** objects, together with a text field named **SearchString**. If the customer searches for products by name, number, or model, the MVC framework will populate this field with the search text provided by the customer.

**Task 3: Implement a Repository Class that Connects to the Cosmos DB Database**

The application will use the Repository Pattern to implement the data access layer. The repository class will be a generic type that the application will use to retrieve data from the collections in the database.

1. Add a new class to the root folder of the project, named **ProductsRepository.cs**.
2. In the **ProductsRepository.cs** file, change the **ProductsRepository** class to a static class that takes a single type parameter, **T**; at runtime, you will create different instances of this repository to handle the **ProductCatalog** and **ProductCategories** collections:
3. public static class ProductsRepository<T>
4. {

}

1. Add the *using* directives shown below to the list at the top of the file:
2. using Microsoft.Azure.Documents.Client;
3. using Microsoft.Azure.Documents.Linq;
4. using System.Configuration;
5. using System.Linq.Expressions;

using System.Threading.Tasks;

1. Add the following static fields to the **ProductsRepository<T>** class. You will use these fields to hold the parameters and objects required to connect to the database and collection:
2. private static string endpointUrl;
3. private static string primaryKey;
4. private static string database;
5. private static string collection;

private static DocumentClient client;

1. Add the static **Initialize** method shown below to the **ProductsRepository<T>** class. This method initializes the fields just defined using settings from the **Web.config** file, and creates a **DocumentClient** object that this class will use to connect to the database:
2. public static void Initialize(string coll)
3. {
4. endpointUrl = ConfigurationManager.AppSettings["EndpointUrl"];
5. primaryKey = ConfigurationManager.AppSettings["PrimaryKey"];
6. database = ConfigurationManager.AppSettings["Database"];
7. collection = coll;
8. client = new DocumentClient(new Uri(endpointUrl), primaryKey);

}

1. Add the following method to the **ProductsRepository<T>** class. This method creates and runs a query that fetches all documents from a collection. The documents are retrieved in list, and the list is returned by the method:
2. public static async Task<IEnumerable<T>> GetAllItemsAsync()
3. {
4. // Find all documents in the collection
5. IDocumentQuery<T> query = client.CreateDocumentQuery<T>(
6. UriFactory.CreateDocumentCollectionUri(database, collection))
7. .AsDocumentQuery();
8. // Return the documents as a list
9. List<T> results = new List<T>();
10. while (query.HasMoreResults)
11. {
12. results.AddRange(await query.ExecuteNextAsync<T>());
13. }
14. return results;

}

1. Add the method shown below to the **ProductsRepository<T>** class. This method creates and runs a query that fetches only those documents that match the expression passed in as the parameter. As before, the documents are retrieved in list, and the list is returned by the method:
2. public static async Task<IEnumerable<T>> GetItemsAsync(Expression<Func<T, bool>> predicate)
3. {
4. // Find all documents in the collection that match the predicate
5. IDocumentQuery<T> query = client.CreateDocumentQuery<T>(
6. UriFactory.CreateDocumentCollectionUri(database, collection),
7. new FeedOptions { EnableCrossPartitionQuery = true })
8. .Where(predicate)
9. .AsDocumentQuery();
10. // Return the matching documents as a list
11. List<T> results = new List<T>();
12. while (query.HasMoreResults)
13. {
14. results.AddRange(await query.ExecuteNextAsync<T>());
15. }
16. return results;

}

1. Open the **Web.config** file and add the following settings to the **appSettings** section. Replace **<URL>** with the URL of your Cosmos DB account and replace **<PRIMARY KEY>** with the primary key of the account:
2. <configuration>
3. <appSettings>
4. <add key="EndpointUrl" value="<URL>" />
5. <add key="PrimaryKey" value="<PRIMARY KEY>" />
6. <add key="Database" value="SalesData" />
7. <add key="ProductsCollection" value="ProductCatalog" />
8. <add key="CategoriesCollection" value="ProductCategories" />
9. ...
10. </appSettings>
11. ...
12. ...

</configuration>

1. Edit the **global.asax** file for the application and add the following two statements to the end of the **Application\_Start**method. These statements initialize two instances of the repository; one for the **ProductCatalog** collection and the other for the **ProductCategories** collection:
2. protected void Application\_Start()
3. {
4. ...
5. ProductsRepository<Adventure\_Works.Models.Product>.Initialize(ConfigurationManager.AppSettings["ProductsCollection"]);
6. ProductsRepository<Adventure\_Works.Models.ProductCategoryData>.Initialize(ConfigurationManager.AppSettings["CategoriesCollection"]);

}

1. Add the following **using** directive to the list of the top of the **global.asax** file:

using System.Configuration;

**Task 4: Create the Products Controller and Initial View**

The application will use the **Products** controller to manage the interactions between the customer and the database. The controller will present data to the customer using a series of views.

1. Add a controller to the **Controllers** folder. Select the **MVC 5 Controller - Empty** template, and name the controller **ProductsController**.
2. Add the following **using** directives to the list at the top of the **ProductsController.cs** file:
3. using Adventure\_Works.Models;

using System.Threading.Tasks;

1. In the **ProductsController** class, create the **InitializeCategoryDataAsync** method shown below:
2. private async Task<IEnumerable<ProductCategoryData>> InitializeCategoryDataAsync()
3. {
4. // Retrieve the list of categories and subcategories from the document database
5. var categoryData = await ProductsRepository<ProductCategoryData>.GetAllItemsAsync();
6. Session["categoryData"] = categoryData;
7. return categoryData;

}

This method uses the **ProductCategoryData** repository to fetch all category (and subcategory) information, and then caches it locally in session memory. The volume of data in this collection is small and can easily be retained in memory. It is also very slow moving (category and subcategory information rarely changes). Caching the data in this way means that the same data does not have to be fetched repeatedly in the user's session.

1. Add the following method to the **ProductsController** class:
2. private async Task<ProductCategoryViewModel> InitializeCategoriesAsync()
3. {
4. // Construct the ProductCategoryViewModel data using the data retrieved from the document database
5. var categoryData = Session["categoryData"] as IEnumerable<ProductCategoryData> ?? await ProductsRepository<ProductCategoryData>.GetAllItemsAsync();
6. var distinctCategories = (from c in categoryData select c.Category).Distinct();
7. var selectableCategories = new ProductCategoryViewModel { Categories = from c in distinctCategories select new SelectListItem { Text = c, Value = c } };
8. Session["selectableCategories"] = selectableCategories;
9. return selectableCategories;

}

This code takes the category data and uses it to populate the **ProductCategoryViewModel** that the MVC framework will use to display product categories to the user. Again, this data is cached in the session. Note that this code uses a LINQ query that eliminates duplicate entries for the same category (each category is repeated for every subcategory that it contains in the **ProductCategories** collection in the Cosmos DB database).

1. Add the method shown below to the **ProductsController** class. This method populates the **ProductSubcategoryViewModel** object used by the MVC framework to display subcategory information:
2. private async Task<ProductSubcategoryViewModel> InitializeSubCategoriesAsync()
3. {
4. // Construct the ProductSubcategoryViewModel data using the data retrieved from the document database
5. var categoryData = Session["categoryData"] as IEnumerable<ProductCategoryData> ?? await ProductsRepository<ProductCategoryData>.GetAllItemsAsync();
6. var subcategoryData = (from c in categoryData select c.Subcategory);
7. var selectableSubcategories = new ProductSubcategoryViewModel { Subcategories = from s in subcategoryData select new SelectListItem { Text = s, Value = s } };
8. Session["selectableSubcategories"] = selectableSubcategories;
9. return selectableSubcategories;

}

1. Delete the following method from the **ProductsController** class:
2. // GET: Products
3. public ActionResult Index()
4. {
5. return View();

}

1. Add the **FindProductsAsync** method shown below to the **ProductsController** class:
2. // Initial, default action for the form
3. [ActionName("FindProducts")]
4. [HttpGet]
5. public async Task<ActionResult> FindProductsAsync()
6. {
7. // Build and display an initial view model containing just the list of categories and subcategories (don't display all products by default as there could be a lot of them)
8. return View(new ProductViewModel
9. {
10. Products = new List<Product>(),
11. SelectableCategories = Session["selectableCategories"] as ProductCategoryViewModel ?? await InitializeCategoriesAsync(),
12. SelectableSubcategories = Session["selectableSubcategories"] as ProductSubcategoryViewModel ?? await InitializeSubCategoriesAsync()
13. });

}

This method creates a **ProductsViewModel** object, populating the **SelectableCategories** and **SelectableSubcategories**fields with the data retrieved and cached by the code shown in the previous steps. Note that the list of products in the view model is left empty initially; there could be a lot of products in the database, and some aspects (such as the number in stock) can change frequently, so populating and caching this list would not be beneficial at this time. After constructing the view model, this method creates a view that is used to display it.

1. Add a view to the **Products** folder in the **Views** folder. Set the name to **FindProducts** (this name should match that of the method in the **ProductsController** that creates it). Select the **Empty (without model)** template, select the **Use a layout page** option, and specify the **~/Views/Shared/\_Layout.cshtml** layout page.
2. In the **FindProducts.cshtml** file, add the following code to the top of the file. This code specifies the type of the object that will provide the data to display in this view:

@model Adventure\_Works.Models.ProductViewModel

1. Change the page heading as follows:

<h1>Product Catalog</h1>

1. Add the following HTML markup and Razor code immediately after the page heading:
2. <div class="panel panel-default">
3. <div class="panel panel-body">
4. @using (Html.BeginForm("FindProducts", "Products", FormMethod.Post, new { role = "form" }))
5. {
6. <div class="form-group">
7. <h2>Categories</h2>
8. @Html.DropDownListFor(c => c.SelectableCategories.CategoryName, new SelectList(Model.SelectableCategories.Categories, "Text", "Value"), new { @class = "form-control" })
9. </div>
10. <p>
11. <button type="submit" class="btn btn-primary">Search By Category</button>
12. </p>
13. }
14. @using (Html.BeginForm("FindProducts", "Products", FormMethod.Post, new { role = "form" }))
15. {
16. <div class="form-group">
17. <h2>Subcategories</h2>
18. @Html.DropDownListFor(s => s.SelectableSubcategories.SubcategoryName, new SelectList(Model.SelectableSubcategories.Subcategories, "Text", "Value"), new { @class = "form-control" })
19. </div>
20. <p>
21. <button type="submit" class="btn btn-primary">Search By Subcategory</button>
22. </p>
23. }
24. @using (Html.BeginForm("FindProducts", "Products", FormMethod.Post, new { role = "form" }))
25. {
26. <div class="form-group">
27. <h2>Product Name, Model Name, or Product Number</h2>
28. @Html.TextBoxFor(t => t.SearchString, new { @class = "form-control" })
29. </div>
30. <p>
31. <button type="submit" class="btn btn-primary">Search By Product Name, Model Name, or Product Number</button>
32. </p>
33. }
34. <!--TODO: Table showing matching products/-->
35. </div>

</div>

This code creates three HTML forms:

* 1. The first form uses the **Html.DropDownList** function to create a drop-down list containing the categories. When the user selects an item from this list, the MVC runtime will fill the **CategoryName** field of the view model with the name of the category.
  2. The second form is similar, except that it displays a drop-down list of subcategories. The MVC runtime will store the selected subcategory name in the **SubcategoryName** field of the view model.
  3. The third form just contains a text box into which the user can enter a product name, number, or model. The value input by the user is stored in the **SearchString** field of the view model.

In all cases the form also displays a button that sends an HTTP Post request back to the controller. The Post request is routed to the **FindProducts** method in the controller.

1. In Solution Explorer, right-click the **Adventure-Works** project (not the solution) and then click **Properties**.
2. On the **Web** tab under **Start Action**, select **Specific Page**, but leave the page name empty. Save the file.
3. In the **App\_Start** folder, edit the **RouteConfig.cs** file. In the **RegisterRoute** method, add the following statement before the code that specifies the **Default** route. This code causes the **ProductsController** to run by default when the user connects to the web site:
4. routes.MapRoute(
5. name: "Home",
6. url: "",
7. defaults: new { controller = "Products", action = "FindProducts" }

);

1. Edit the **\_Layout.cshtml** file in the **Views/Shared** folder. In this file:
   1. Set the **<title>** to **Adventure-Works Cycles**.
   2. Set the title in **Html.ActionLink** action in the navigation bar header to **Adventure-Works Product Catalog**.
   3. Set the **<footer>** message to **@DateTime.Now.Year - Adventure-Works Cycles**.
   4. The code below shows the complete file after these changes are made:
2. <!DOCTYPE html>
3. <html>
4. <head>
5. <meta charset="utf-8" />
6. <meta name="viewport" content="width=device-width, initial-scale=1.0">
7. <title>@ViewBag.Title Adventure-Works Cycles</title>
8. @Styles.Render("~/Content/css")
9. @Scripts.Render("~/bundles/modernizr")
10. </head>
11. <body>
12. <div class="navbar navbar-inverse navbar-fixed-top">
13. <div class="container">
14. <div class="navbar-header">
15. <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
16. <span class="icon-bar"></span>
17. <span class="icon-bar"></span>
18. <span class="icon-bar"></span>
19. </button>
20. @Html.ActionLink("Adventure-Works Product Catalog", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
21. </div>
22. <div class="navbar-collapse collapse">
23. <ul class="nav navbar-nav">
24. <li>@Html.ActionLink("Home", "Index", "Home")</li>
25. <li>@Html.ActionLink("Catalog", "FindProducts", "Products")</li>
26. <li>@Html.ActionLink("About", "About", "Home")</li>
27. <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
28. </ul>
29. </div>
30. </div>
31. </div>
32. <div class="container body-content">
33. @RenderBody()
34. <hr />
35. <footer>
36. <p>&copy; @DateTime.Now.Year - Adventure-Works Cycles</p>
37. </footer>
38. </div>
39. @Scripts.Render("~/bundles/jquery")
40. @Scripts.Render("~/bundles/bootstrap")
41. @RenderSection("scripts", required: false)
42. </body>

</html>

1. Build and run the web application. Verify that the **Product Catalog** page appears. The **Categories** and **Subcategories**lists should be populated (click the drop-down lists to see the data). Note that if you click any of the buttons, you will currently receive an error because you have not implemented the code that handles the POST request yet. Close the web application.

**Task 5: Extend the Products Controller to Fetch and Display Product Data**

1. Edit the **ProductsController** class and add the following overload for the **FindProductsAsync** method:
2. // Method called as POST request from the various search buttons on the form
3. [ActionName("FindProducts")]
4. [HttpPost]
5. public async Task<ActionResult> FindProductsAsync(ProductViewModel productViewModel)
6. {
7. // Construct a view model that matches the search criteria specified by the user
8. IEnumerable<Product> products = null;
9. if (productViewModel.SelectableCategories != null)
10. {
11. // User is searching by category
12. products = await ProductsRepository<Product>.GetItemsAsync(p => p.ProductCategory.Category == productViewModel.SelectableCategories.CategoryName);
13. }
14. else if (productViewModel.SelectableSubcategories != null)
15. {
16. // User is searching by subcategory
17. products = await ProductsRepository<Product>.GetItemsAsync(p => p.ProductCategory.Subcategory == productViewModel.SelectableSubcategories.SubcategoryName);
18. }
19. else
20. {
21. // User is searching by product name, model name, or product number
22. products = await ProductsRepository<Product>.GetItemsAsync(p =>
23. p.ProductName.Contains(productViewModel.SearchString) ||
24. p.Model.Contains(productViewModel.SearchString) ||
25. p.ProductNumber.Contains(productViewModel.SearchString));
26. }
27. // Construct a new view model containing the results and display it
28. return View(new ProductViewModel
29. {
30. Products = products,
31. SelectableCategories = Session["selectableCategories"] as ProductCategoryViewModel ?? await InitializeCategoriesAsync(),
32. SelectableSubcategories = Session["selectableSubcategories"] as ProductSubcategoryViewModel ?? await InitializeSubCategoriesAsync()
33. });

}

This version of the method responds to the POST request from the view. The method examines the values in the **SelectableCategories**, **SelectableSubcategories**, and **SearchString** fields in the view model (only one of them will contain any data, depending on which button the user clicked in the view). The method then uses this information to populate the **Products** list in the view model with a list of products in the specified category or subcategory, or that match the name, number, or model of the product. Finally, the method constructs a new instance of the view model with this data which it sends back to the view for display.

1. Edit the **FindProducts.cshtml** file. Replace the **<!--TODO: Table showing matching products/-->** markup in the view with the following code:
2. <table class="table">
3. <thead>
4. <tr>
5. <td>Product Number</td>
6. <td>Product Name</td>
7. <td>Model</td>
8. <td></td>
9. </tr>
10. </thead>
11. @foreach (var item in Model.Products)
12. {
13. <tr>
14. <td>
15. @Html.DisplayFor(modelItem => item.ProductNumber)
16. </td>
17. <td>
18. @Html.DisplayFor(modelItem => item.ProductName)
19. </td>
20. <td>
21. @Html.DisplayFor(modelItem => item.Model)
22. </td>
23. <td>
24. @Html.ActionLink("Details", "GetProductDetails", new { productID = item.ProductID })
25. </td>
26. </tr>
27. }

</table>

This code creates an HTML table that displays the data in the Products list in the view model. If this list is empty, no data is displayed. The table also includes a link named **Details**. The intention is that if the user clicks this link, they will be presented with the full details of the selected product (you will implement this functionality shortly).

1. Build and run the web application.
   * From the **Categories** list, pick a category, and then click **Search By Category**. The view should now display all products that match that category.
   * In the **Subcategories** list, select a subcategory, and then click **Search By Subcategory**. This time the view should display only those products in that subcategory.
   * In the **Product Name, Model Name, or Product Number** text box, type **Wash**, and then click **Search By Product Name, Model Name, or Product Number**. You should now be presented with all products that contain the text **Wash** in the product number, name, or model. Note that this search is case-sensitive.
   * If you click the **Details** link for a product, the application will currently report an error because you haven't implemented the **GetProductDetails** method yet.
2. Close the web application and return to Visual Studio.
3. Add the **GetProductDetailsAsync** method shown below to the **ProductsController** class. This method takes the ID of he product selected by the customer when they click the **Details** link, uses the repository to fetch the details of the specified product, and then displays it in another view:
4. // Method called to display the details of the specified product
5. [ActionName("GetProductDetails")]
6. [HttpGet]
7. public async Task<ActionResult> GetProductDetailsAsync(string productID)
8. {
9. // Find the data for the specified product
10. var products = await ProductsRepository<Product>.GetItemsAsync(p => p.ProductID == productID);
11. // Display the details of the product
12. return View(products.First());

}

1. Add a view to the **View/Products** folder. Name the view **GetProductDetails**. Select the template **Empty (without model)**. Select **Use a layout page**, and specify the page **~/Views/Shared/\_Layout.cshtml**.
2. Modify the Razor code in the view; specify **Adventure\_Works.Models.Product** as the model, and change the title and page heading as shown below:
3. @model Adventure\_Works.Models.Product
4. @{
5. ViewBag.Title = "Products";
6. Layout = "~/Views/Shared/\_Layout.cshtml";
7. }

<h1>Product Details</h1>

1. Add the following HTML markup below the page heading:
2. <div class="panel panel-default">
3. <div class="panel panel-body">
4. <div class="form-group">
5. <table class="table">
6. <tr>
7. <td>Product Number: </td>
8. <td>@Html.DisplayTextFor(p => p.ProductNumber)</td>
9. </tr>
10. <tr>
11. <td>Product Name: </td>
12. <td>@Html.DisplayTextFor(p => p.ProductName)</td>
13. </tr>
14. <tr>
15. <td>Model: </td>
16. <td>@Html.DisplayTextFor(p => p.Model)</td>
17. </tr>
18. <tr>
19. <td>Description: </td>
20. <td>@Html.DisplayTextFor(p => p.Description)</td>
21. </tr>
22. <tr>
23. <td>Price: </td>
24. @if (Model.ListPrice == 0)
25. {
26. <td><p>Contact Adventure-Works for price information</p></td>
27. }
28. else
29. {
30. <td>@Html.DisplayTextFor(p => p.ListPrice)</td>
31. }
32. </tr>
33. <tr>
34. <td>Color: </td>
35. <td>@Html.DisplayTextFor(p => p.Color)</td>
36. </tr>
37. <tr>
38. <td>Size: </td>
39. <td>@Html.DisplayTextFor(p => p.Size)</td>
40. </tr>
41. <tr>
42. <td>Weight: </td>
43. <td>@Html.DisplayTextFor(p => p.Weight)</td>
44. </tr>
45. <tr>
46. <td>In Stock: </td>
47. <td>@Html.DisplayTextFor(p => p.QuantityInStock)</td>
48. </tr>
49. @if (Model.Documentation.Document.Length > 0)
50. {
51. <tr>
52. <td>Documents: </td>
53. <td>
54. @Html.DisplayTextFor(p => p.Documentation)
55. <a href="@Model.Documentation.Document">Click to download</a>
56. </td>
57. </tr>
58. }
59. <tr>
60. <td>Image: </td>
61. <td>
62. <img src="@Model.Images.LargePhoto" />
63. </td>
64. </tr>
65. </table>
66. </div>
67. </div>

</div>

This code displays the details of the product in a tabular layout. Note that the link for product documentation is specified as an anchor (the user can click this link to download the documentation), and the image is displayed using an **<img>** tag. In both cases, the data is downloaded from blob storage.

1. Build and run the web app. It should function as before, but now you should be able to click the **Details** link for a product to view the data for that product.
2. Close the web app.

**Result**: At the end of this exercise, you should have created an application that you can use to query product information.

**Exercise 4: Maintaining Stock Levels in the Product Catalog**

**Exercise 4 Scenario**

You will extend the web app to enable customers to buy items. The data for each product includes the number in stock, and this value should be updated as purchases occur. As is common in many online retail systems, the stock level displayed could become out of date while the customer is browsing products. If there is insufficient stock available when the customer commits to buying an item, it will be placed on back order instead and the customer notified. This will involve creating a new **backorder** document and saving it in a collection in Cosmos DB for a separate back-orders system to process.

**Note**: The back-orders processing system is not implemented in this lab.

The main tasks for this exercise are as follows:

1. Create a collection and repository for storing back orders.
2. Extend the repository class to enable document creation and update.
3. Add **buy** functionality to the product details view.
4. Close down the lab.

**Task 1: Create a Collection and Repository for Storing Back Orders**

1. Using the Azure portal, add a new collection named **BackOrders** to the **SalesData** database in your Cosmos DB account. Partition the data by **productid**, and set the throughput to **1000** RU/s.
2. Using Visual Studio, open the **Adventure-Works** solution in the **E:\Labfiles\Lab02\Starter\Adventure-Works-Ex4**folder. This is a copy of the completed web app from exercise 3.
3. Edit the **Web.config** file, and add the following setting to the list of app settings:
4. <appSettings>
5. ...
6. <add key="BackOrderCollection" value="BackOrders"/>
7. ...

</appSettings>

1. Edit the **Product.cs** file in the **Models** folder, and add the **ProductBackOrder** class shown below to this file. This class represents the document structure for a back order:
2. public class ProductBackOrder
3. {
4. [JsonProperty("id")]
5. public string BackOrderID { get; set; }
6. [JsonProperty("productid")]
7. public string ProductID { get; set; }
8. [JsonProperty("productname")]
9. public string ProductName { get; set; }
10. [JsonProperty("productnumber")]
11. public string ProductNumber { get; set; }
12. [JsonProperty("numberonbackorder")]
13. public int NumberOnBackOrder { get; set; }
14. [JsonProperty("backorderdate")]
15. public DateTime BackOrderDate { get; set; }

}

1. Open the **global.asax** file, and add the following statement to the end of the **Application\_Start** method, to create a repository for handling **ProductBackOrder** documents:
2. protected void Application\_Start()
3. {
4. ...
5. ...
6. ProductsRepository<Adventure\_Works.Models.ProductBackOrder>.Initialize(ConfigurationManager.AppSettings["BackOrderCollection"]);

}

**Task 2: Extend the Repository Class to Enable Document Creation and Update**

1. Edit the **ProductsRepository.cs** file, and add the following **using** directives to the list at the top of the file:
2. using System.Net;

using Microsoft.Azure.Documents;

1. Add the **CreateItem** method shown below to the end of the **ProductsRepository<T>** class. This method uses the SQL API to add the document represented by the **item** parameter to the collection. It returns a Boolean value indicating whether the create operation was successful:
2. public static async Task<bool> CreateItem(T item)
3. {
4. // Add the new document to the collection
5. Uri collectionUri = UriFactory.CreateDocumentCollectionUri(database, collection);
6. var response = await client.CreateDocumentAsync(collectionUri, item);
7. // Examine the status of the response, to determine whether the document was created successfully or not
8. return response.StatusCode == HttpStatusCode.Created;

}

1. Add the **IncrementNumericValueInDocument** method shown below to the **ProductsRepository<T>** class:
2. public static async Task<int> IncrementNumericValueInDocumentAsync(string docID, string property, int value, string partitionKey = null)
3. {
4. // TODO: Fetch the doc to be updated
5. // TODO: Change the value of the specified property
6. // TODO: Attempt to write the modified document back to the database
7. // If another user modifies the same doc at the same time, this code will throw an exception back to the caller where it should be handled
8. // TODO: Return the new value of the property that was updated

}

The purpose of this method is to increment the numeric field specified by the **property** parameter in a document identified by the **docID** parameter by the amount specified in the **value** parameter. The optional **partionKey**parameter is used to locate the partition in which the document resides (this parameter is required if a collection is partitioned).

1. Replace the comment **TODO: Fetch the doc to be updated** with the following block of code. These statements ensure that the most recent version of the product document is retrieved; the document might have been updated while the customer was browsing:
2. Uri docUri = UriFactory.CreateDocumentUri(database, collection, docID);
3. var options = partitionKey == null ? null : new RequestOptions
4. {
5. PartitionKey = new PartitionKey(partitionKey)
6. };
7. var response = await client.ReadDocumentAsync(docUri, options);

var doc = response.Resource;

1. Replace the comment **TODO: Change the value of the specified property** with these statements. This code uses the generic **GetPropertyValue** method of the SQL API to read the value of the named property from the document. The **SetPropertyValue** method overwrites the value of the property in the document:
2. var propVal = doc.GetPropertyValue<int>(property);
3. propVal += value;

doc.SetPropertyValue(property, propVal);

1. Replace the comment **TODO: Attempt to write the modified document back to the database** with the following code.
2. options = new RequestOptions
3. {
4. AccessCondition = new AccessCondition
5. {
6. Condition = doc.ETag,
7. Type = AccessConditionType.IfMatch
8. }
9. };

response = await client.ReplaceDocumentAsync(doc.SelfLink, doc, options);

There is a chance that another user could have updated the same document at the same time. This code uses the **ETag** property of the document to check for concurrent update, and throws an exception if the document has been modified. The client application calling this method should be prepared to handle the exception; for example, it could re-query the database and display the latest version of the data, and then enable the user to repeat the operation if necessary:

1. Replace the comment **TODO: Return the new value of the property that was updated** with the following statement:

return propVal;

**Task 3: Add Buy Functionality to the Product Details View**

1. In the **GetProductDetails.cshtml** file in the **View/Products** folder, add the following HTML markup immediately after the closing **</table>** tag:
2. @using (Html.BeginForm("BuyProduct", "Products", FormMethod.Post, new { role = "form" }))
3. {
4. <div class="form-group">
5. <p>Number Required:
6. <input name="productID" type="hidden" value="@Model.ProductID" />
7. <input name="subcategory" type="hidden" value="@Model.ProductCategory.Subcategory" />
8. <input name="productName" type="hidden" value="@Model.ProductName" />
9. <input name="productNumber" type="hidden" value="@Model.ProductNumber" />
10. <input id="numRequired" name="numRequired" class="form-control" type="number" value="1" min="1"/></p>
11. <p>
12. <button type="submit" class="btn btn-primary">Purchase</button>
13. </p>
14. </div>

}

This markup creates an HTML form that enables the user to enter the number of items required. The **Purchase** button triggers the **BuyProduct** method in the **ProductsController** (you will write this method shortly). The parameters passed to the **BuyProduct** method consist of the value in the **numRequired** field, together with the contents of the hidden fields containing the product id, category, subcategory, and number.

1. Add the following style after the last closing **</div>** tag in the file. This style sets the display width of the **numRequired** field on the form:
2. <style>
3. #numRequired {
4. width: 6em;
5. }

</style>

1. Open the **ProductsController.cs** file in the **Controllers** folder. Add the following method named **BuyProductAsync** to the **ProductsController** class:
2. // Method called when the user wishes to purchase a product
3. [ActionName("BuyProduct")]
4. [HttpPost]
5. public async Task<ActionResult> BuyProductAsync(string productID, string subcategory, string productName, string productNumber, int numRequired)
6. {
7. // Use the repository to update the database and return the new stock level
8. int newStockLevel = await ProductsRepository<Product>.IncrementNumericValueInDocumentAsync(productID, "quantityinstock", -numRequired, subcategory);
9. // If the new stock level is now negative, undo the update and place a back order instead (Adventure-Works currently doesn't allow split orders)
10. if (newStockLevel < 0)
11. {
12. await ProductsRepository<Product>.IncrementNumericValueInDocumentAsync(productID, "quantityinstock", numRequired, subcategory);
13. var backOrder = new ProductBackOrder
14. {
15. BackOrderID = Guid.NewGuid().ToString(),
16. ProductID = productID,
17. ProductName = productName,
18. ProductNumber = productNumber,
19. NumberOnBackOrder = numRequired,
20. BackOrderDate = DateTime.UtcNow
21. };
22. await ProductsRepository<ProductBackOrder>.CreateItem(backOrder);
23. }
24. return View(newStockLevel);

}

This method runs when the customer clicks **Purchase**. It uses the products repository to update the stock level (this is a decrement operation rather than increment, so the amount specified is converted to a negative value). If the quantity left in stock as a result of this operation is negative, then there is insufficient available to satisfy the order. In this case, the stock level is reinstated and a back order placed instead (the web application does not currently support orders split across multiple deliveries). The results of the request are displayed using another view.

1. Add another view to the **Views/Products** folder. Name the view **BuyProduct**. Select the template **Empty (without model)**. Select **Use a layout page**, and specify the page **~/Views/Shared/\_Layout.cshtml**.
2. Replace the markup in the view with the following code. In this view, the **model** data consists of the new stock level. If this value is negative there was insufficient stock available so the message **Backorder created** appears, otherwise the message **Order placed** is displayed:
3. @model int
4. @{
5. ViewBag.Title = "Products";
6. Layout = "~/Views/Shared/\_Layout.cshtml";
7. }
8. <h1>Product Details</h1>
9. <div class="panel panel-default">
10. <div class="panel panel-body">
11. @if (Model < 0)
12. {
13. <p>Backorder created</p>
14. } else
15. {
16. <p>Order placed</p>
17. }
18. </div>

</div>

1. Build and run the application. Display the details for a product and make a purchase. Verify that the **Order placed**message appears. View the product again, and verify that the number in stock has changed.
2. Attempt to buy 5000 of the same product. This time the message **Backorder created** should be displayed. View the product again, and verify that the number in stock hasn't changed this time, then close the web app.
3. Use the Azure Portal Data Explorer to examine the **BackOrders** collection in the **SalesData** database. Verify that a document has been added that specifies a backorder of 5000 of the product you selected.

**Task 4: Lab close down**

* To reduce your costs delete the **20777\_Mod02** resource group containing your Cosmos DB database and storage account.

**Result**: In this lab, you have used what you have learned in this lesson to design a document schema, load data into a collection, and to query, modify, and delete documents in a collection.

**Lab: Designing and Implementing a Document Database**

**Exercise 1: Design the Document Structure for the Product Catalog**

**Task 1: Prepare the Environment**

1. Ensure that the **MT17B-WS2016-NAT** and **20777A-LON-DEV** virtual machines are running, and then log on to **20777A-LON-DEV** as **Administrator** with the password **Pa55w.rd**.
2. On the Windows toolbar, click **File Explorer**.
3. In File Explorer, navigate to **E:\Labfiles\Lab02\Starter**, right-click **Setup.cmd**, and then click **Run as administrator**.
4. Wait for the script to finish.
5. On the Windows toolbar, click **Internet Explorer**.
6. In Internet Explorer, go to [**http://portal.azure.com**](http://portal.azure.com/), and sign in using the Microsoft account that is associated with your Azure Learning Pass subscription.
7. In the Azure Portal, click **+ Create a resource**.
8. On the **New** blade, in the search box, type **storage account**, and then press Enter.
9. On the **Everything** blade, click **Storage account - blob, file, table, queue**.
10. On the **Storage account - blob, file, table, queue** blade, click **Create**.
11. On the **Create storage account** blade, under the **Resource group** box, click **Create new**, type **20777\_Mod02**, and then click **OK**.
12. In the **Storage account name** box, type **20777blob<*your name><the day*>**, for example, **20777blobjohn31**.
13. In the **Location** drop-down list, click the region closest to your current location.
14. In the **Account kind** list, click **BlobStorage**.
15. In the **Replication** list, click **Locally-redundant storage (LRS)**.
16. Click **Review + create**, and then click **Create**.
17. In the Azure Portal, click **All resources**, and then click **20777blob<*your name><the day*>**.
18. On the **20777blob<*your name><the day*>** blade, under **Blob Service**, click **Blobs**.
19. On the **Blobs** blade, click **+ Container**.
20. In the **New container** dialog box, in the **Name** box, type **20777-mod2-blobs**.
21. In the **Public access level** drop-down list, click **Blob (anonymous read access for blobs only)**, and then click **OK**.
22. On the **20777blob<*your name><the day*>** blade, under **Settings**, click **Access keys**.
23. Under **key1**, make a note of the **Connection string** value.
24. In the left pane of the Azure portal, click **+ Create a resource**.
25. On the **New** blade, in the search box, type **Cosmos DB**, and then press Enter.
26. On the **Everything** blade, click **Azure Cosmos DB**, and then click **Create**.
27. On the **Create Azure Cosmos DB Account** blade, in the **Resource Group** drop-down list, click **20777\_Mod02**.
28. In the **Account name** box, type **20777-sql-<*your name>-<the day*>**, for example, **20777-sql-john-31**.
29. In the **API** drop-down list, click **Core (SQL)**.
30. In the **Location** drop-down list, click the region closest to your current location.
31. Click **Review + create**, and then click **Create**.
32. Wait for the Azure Cosmos DB to be created—this could take a few minutes.
33. In the Azure Portal, click **All resources**, and then click **20777-sql-<*your name>-<the day*>**.
34. On the **20777-sql-<*your name>-<the day*>** blade, click **Data Explorer**.
35. In the **SQL API** pane, click **New Database**.
36. On the **New Database** blade, in the **Database id** box, type **SalesData**, and then click **OK**.
37. On the **20777a-sql-<*your name>-<the day*>** blade, under **Settings**, click **Keys**.
38. Make a note of the **URI**, and **PRIMARY KEY** values.

**Task 2: Analyze the requirements**

There are no detailed steps for this task. Read the description for the task in **20777A\_LAB\_02.md**.

**Task 3: Design the Documents and Partitioning Strategy**

1. On the Start menu, click **Visual Studio 2017**.
2. In Visual Studio, on the **Tools** menu, click **Connect to Database**.
3. In the **Choose Data Source** dialog box, click **Microsoft SQL Server Database File**, and then click **Continue**.
4. In the **Add Connection** dialog box, in the **Database file name (new or existing)** box, type **E:\Labfiles\Lab02\Data\AdventureWorks2016\_Data.mdf**, and then click **OK**.
5. In Server Explorer, right-click **AdventureWorks2016\_Data.mdf**, and then click **New Query**.
6. On the **SQLQuery1.sql** tab, type the following code:
7. SELECT c.Name AS category, s.Name AS subcategory
8. FROM Production.ProductCategory c
9. JOIN Production.ProductSubcategory s

ON s.ProductCategoryID = c.ProductCategoryID

1. On the **SQL** menu, click **Execute**, and then review the results of the query.
2. In the **SQLQuery1.sql** tab, delete the existing query, and then type the following code:
3. SELECT p.ProductID, p.Name, p.ProductNumber, p.Color, p.ListPrice, p.Size, pum.Name As SizeUnit,
4. p.Weight, pw.Name As WeightUnit, psc.Name As SubcategoryName, pc.Name As CategoryName, i.Quantity,
5. d.Title, d.DocumentSummary, d.Document, d.FileName AS DocumentFileName, pm.Name AS ModelName,ill.Diagram,
6. ph.ThumbnailPhoto, ph.ThumbnailPhotoFileName, ph.LargePhoto, ph.LargePhotoFileName, pds.Description
7. FROM Production.Product p
8. LEFT JOIN Production.ProductSubcategory psc
9. ON p.ProductSubCategoryID = psc.ProductSubcategoryID
10. LEFT JOIN Production.ProductCategory pc
11. ON psc.ProductCategoryID = pc.ProductCategoryID
12. LEFT JOIN(SELECT ProductID, SUM(Quantity) AS Quantity FROM Production.ProductInventory GROUP BY ProductID) AS i
13. ON p.ProductID = i.ProductID
14. LEFT JOIN Production.UnitMeasure pum
15. ON p.SizeUnitMeasureCode = pum.UnitMeasureCode
16. LEFT JOIN Production.UnitMeasure pw
17. ON p.WeightUnitMeasureCode = pw.UnitMeasureCode
18. LEFT JOIN Production.ProductDocument pd
19. ON p.ProductID = pd.ProductID
20. LEFT JOIN Production.Document d
21. ON pd.DocumentNode = d.DocumentNode
22. LEFT JOIN Production.ProductModel pm
23. ON p.ProductModelID = pm.ProductModelID
24. LEFT JOIN Production.ProductModelIllustration pmi
25. ON pm.ProductModelID = pmi.ProductModelID
26. LEFT JOIN Production.Illustration ill
27. ON pmi.IllustrationID = ill.IllustrationID
28. LEFT JOIN Production.ProductProductPhoto ppp
29. ON P.ProductID = ppp.ProductID
30. LEFT JOIN Production.ProductPhoto ph
31. ON ppp.ProductPhotoID = ph.ProductPhotoID
32. LEFT JOIN Production.ProductModelProductDescriptionCulture pmdc
33. ON pm.ProductModelID = pmdc.ProductModelID AND pmdc.CultureID = 'en'
34. LEFT JOIN Production.ProductDescription pds

ON pmdc.ProductDescriptionID = pds.ProductDescriptionID

1. On the **SQL** menu, click **Execute**, and then review the results of the query.

**Question**: What different approaches to the document design did you consider; what are their advantages and disadvantages?

**Answer**: The answer will depend on students' experience. You could create a document for each category containing the subcategories as sub-documents, and products as sub-sub-documents within each subcategory. However, this is probably not the best approach as the number of products in each category/subcategory is unbounded. The result would be a small number of very large documents and a solution that doesn't scale well.

An alternative approach is to create a document for each product, and store category/subcategory information as properties within each product. This solution scales better. But you also need to consider how applications will let a user select category and subcategory information for performing searches. For example, if the user wants to search for products by subcategory, it would be useful to be able to present the user with a list of subcategories from which to make a selection. If this data is duplicated across many product documents, generating a list of categories and subcategories could be time consuming.

**Question**: Will you store all the documents in one collection, or use more than one collection?

**Answer**: The answer will depend on students' experience. The most optimal solution is to create two collections; one for products and another for category/subcategory information. Products can either be linked to the appropriate category/subcategory document - this would enable category/subcategory information to be easily updated as it is only held in one place -, or the category/subcategory data can be copied to each document - this would result in faster queries as each time a document is retrieved you don't need to perform an additional retrieval of category/subcategory information. Given that categories and subcategories don't change very often, and that the system must support fast querying, the best solution is to copy the data into each product.

**Question**: The product data includes binary information in the form of Word documents and image files. How will you accommodate these files in Cosmos DB?

**Answer**: The answer will depend on students' experience. This data is best held either as attachments in the Cosmos DB database, or in separate blob storage (you don't necessarily want to download this information every time you query a product). There is a limit on the amount of data you can hold as attachments in a collection, and there could be a large number of images and other objects referenced by products in the catalog, so the preferred solution is to store this data in Azure blob storage and add a URL that references the appropriate data to each product document.

The products collection could have the following structure:

{

"id": "977",

"productname": "Road-750 Black, 58",

"productnumber": "BK-R19B-58",

"color": "Black",

"listprice": 539.99,

"size": "58 Centimeter",

"weight": "20.79 US pound",

"quantityinstock": 153,

"model": "Road-750",

"description": "Entry level adult bike; offers a comfortable ride cross-country or down the block. Quick-release hubs and rims.",

"productcategory": {

"subcategory": "Road Bikes",

"category": "Bikes"

},

"documentation": {

"documenttitle": "Introduction 1",

"documentsummary": "",

"document": "<link to blob storage>"

},

"images": {

"diagram": "",

"thumbnail": "<link to blob storage>",

"largephoto": "<link to blob storage>"

}

}

Documents in the category/subcategory collection could look like this. There is a document for each subcategory:

{

"category": "Bikes",

"subcategory": "Mountain Bikes",

}

{

"category": "Bikes",

"subcategory": "Road Bikes",

}

1. Identify a partitioning strategy for the product data.

**Question**: Assuming that the collection was going to scale to many millions of products, how would you partition the products collection?

**Answer**: The answer will depend on students' experience. You could partition product data by **ProductNumber**, but this would lead to each product being stored in its own partition. Querying by category or subcategory would lead to many cross-partition queries, and this strategy would be very inefficient.

Partitioning products by category would likely be too coarse a strategy as there are very few categories (4 currently)

Partitioning products by subcategory will create more partitions to help distribute the load, while still reducing the likelihood of performing cross-partition queries.

**Note**: partitioning by subcategory could result in data skew (there is not an even distribution of products across subcategories, so some partitions will be bigger then others), is this a problem in this scenario? Probably not - the collection is query heavy (updates are far less frequent). Searches that locate products by subcategory will only need to scan a single partition. Using a different partition key could help to distribute updates more evenly, but would impact the response time of queries.

**Question**: Will the product category/subcategory documents benefit from partitioning?

**Answer**: The category/subcategory collection is very small and is unlikely to grow; it doesn't currently require partitioning. This strategy will also ensure that queries that generate a list of categories from the subcategory documents do not have to perform cross-partition searches.

**Task 4: Create a Document Database Using the Cosmos DB SQL API**

1. In Internet Explorer, on the **20777-sql-<*your name>-<the day*>** blade, click **Data Explorer**.
2. In the **SQL API** pane, right-click **SalesData**, and then click **New Collection**.
3. On the **Add Collection** blade, in the **Collection Id** box, type **ProductCatalog**.
4. In the **Partition key** box, type **/productcategory/subcategory**.
5. In the **Throughput (400 - 1,000,000 RU/s)** box, type **1000**.
6. Under **Unique keys**, click **+ Add unique key**.
7. In the **Unique keys** box, type **/productnumber**, and then click **OK**.
8. In the **SQL API** pane, right-click **SalesData**, and then click **New Collection**.
9. On the **Add Collection** blade, in the **Collection Id** box, type **ProductCategories**.
10. In the **Partition key** box, type **/productcategory/subcategory**.
11. In the **Throughput (400 - 1,000,000 RU/s)** box, type **1000**.
12. Under **Unique keys**, click **+ Add unique key**.
13. In the **Unique keys** box, type **/subcategory**, and then click **OK**.

**Result**: At the end of this exercise, you should have created a database and collection to hold the product catalog, and designed the document schema.

**Exercise 2: Import Data into the SalesData Database**

**Task 1: Import Category and Subcategory Data into the Cosmos DB Database**

1. In Visual Studio, on the **SqlQuery1.sql** tab, delete the existing text, and then type the following code:
2. SELECT c.Name AS category, s.Name AS subcategory
3. FROM Production.ProductCategory c
4. JOIN Production.ProductSubcategory s
5. ON s.ProductCategoryID = c.ProductCategoryID

FOR json path

1. On the **SQL** menu, click **Execute**.
2. In the **Results** pane, click the first cell of the result (the text is underlined in blue). This opens a new tab (with a name that starts **JSON\_**) with the results of the query.
3. On the **File** menu, click **Save JSON\_<identifier> As** (**<identifier>** will be a random value).
4. In the **Save File As** dialog box, in the **File name** box, type **E:\Labfiles\Lab02\categories.json**, and then click **Save**.
5. In Internet Explorer, in the **SQL API** pane, expand **SalesData**, click **ProductCategories**, and then click **Upload**.
6. On the **Upload Documents** blade, click browse (the folder icon).
7. In the **Choose** **File to Upload** dialog box, in the **File name** box, type **E:\Labfiles\Lab02\categories.json**, and then click **Open**.
8. On the **Upload Documents** blade, click **Upload**.
9. Wait for the upload to complete; the result message will report that **38 documents created**.
10. In the **SQL API** pane, under **ProductCategories**, click **Documents**.
11. On the **Documents** tab, click a document to verify its contents.

**Task 2: Import Product Data into the Cosmos DB Database**

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, in the **File name** box, type **E:\Labfiles\Lab02\Starter\MigrateProductData\MigrateProductData.sln**, and then click **Open**.
3. In the **Microsoft Visual Studio** dialog box, prompting to save changes, click **No**.
4. In the **Security Warning for MigrateProductData** dialog box, clear the **Ask me for every project in this solution** check box, click **OK**.
5. In Solution Explorer, double-click **App.config**.
6. In App.config, in the **Value** attribute of the **EndpointUrl** key, paste the **URI** value you noted earlier, replacing the text <**URL**>.
7. In the **Value** attribute of the **PrimaryKey** key, paste the **PRIMARY KEY** value you noted earlier, replacing the text **<PRIMARY KEY>**.
8. In the **Value** attribute of the **BlobStorageConnectionString** key, paste the **Connection string** value you noted earlier, replacing the text **<CONNECTION STRING>**.
9. In the **Value** attribute of the **BlobContainer** key, delete the text **<BLOB CONTAINER NAME>**, and then type **20777-mod2-blobs**.
10. In Solution Explorer, double-click **Product.cs**.
11. In Product.cs, under the comment **TODO: Specify fields for the Product document**, replace:
12. internal class Product
13. {

}

with:

internal class Product

{

[JsonProperty("id")]

internal string ProductID { get; set; }

[JsonProperty("productname")]

internal string ProductName { get; set; }

[JsonProperty("productnumber")]

internal string ProductNumber { get; set; }

[JsonProperty("color")]

internal string Color { get; set; }

[JsonProperty("listprice")]

public decimal ListPrice { get; set; }

[JsonProperty("size")]

internal string Size { get; set; }

[JsonProperty("weight")]

internal string Weight { get; set; }

[JsonProperty("quantityinstock")]

internal int QuantityInStock { get; set; }

[JsonProperty("model")]

internal string Model { get; set; }

[JsonProperty("description")]

internal string Description { get; set; }

[JsonProperty("productcategory")]

internal ProductCategoryData ProductCategory { get; set; }

[JsonProperty("documentation")]

internal ProductDocumentData Documentation { get; set; }

[JsonProperty("images")]

internal ProductImageData Images { get; set; }

}

1. Under the comment **TODO: Specify fields for the ProductCategoryData subdocument**, replace:
2. internal class ProductCategoryData
3. {

}

with:

internal class ProductCategoryData

{

[JsonProperty("subcategory")]

internal string Subcategory { get; set; }

[JsonProperty("category")]

internal string Category { get; set; }

}

1. Under the comment **TODO: Specify fields for the ProductDocumentData subdocument**, replace:
2. internal class ProductDocumentData
3. {

}

with:

internal class ProductDocumentData

{

[JsonProperty("documenttitle")]

internal string DocumentTitle { get; set; }

[JsonProperty("documentsummary")]

internal string DocumentSummary { get; set; }

[JsonProperty("document")]

internal string Document { get; set; }

}

1. Under the comment **TODO: Specify fields for the ProductImageData subdocument**, replace:
2. internal class ProductImageData
3. {

}

with:

internal class ProductImageData

{

[JsonProperty("diagram")]

internal string Diagram { get; set; }

[JsonProperty("thumbnail")]

internal string Thumbnail { get; set; }

[JsonProperty("largephoto")]

internal string LargePhoto { get; set; }

}

1. In Solution Explorer, double-click **Program.cs**.
2. Under the comment **TODO: Create a client to connect to the Cosmos DB database**, type the following code:

this.client = new DocumentClient(new Uri(endpointUrl), primaryKey);

1. Under the comment **TODO: Connect to Azure Blob Storage**, type the following code:
2. if (!CloudStorageAccount.TryParse(this.blobStorageConnectionString, out CloudStorageAccount storageAccount))
3. {
4. Console.WriteLine("Invalid storage connection string");
5. return;
6. }

this.blobClient = storageAccount.CreateCloudBlobClient();

1. Under the comment **TODO: Create a document from the current row retrieved from SQL Server**, type the following code:
2. var productDocument = new Product
3. {
4. ProductID = reader["ProductID"].ToString(),
5. ProductName = reader["Name"].ToString(),
6. ProductNumber = reader["ProductNumber"].ToString(),
7. Color = reader["Color"].ToString(),
8. ListPrice = Decimal.Parse(reader["ListPrice"].ToString()),
9. Size = $"{reader["Size"].ToString()} {reader["SizeUnit"].ToString()}",
10. Weight = $"{reader["Weight"].ToString()} {reader["WeightUnit"].ToString()}",
11. QuantityInStock = Int32.Parse(reader["Quantity"].ToString() == "" ? "0" : reader["Quantity"].ToString()),
12. Model = reader["ModelName"].ToString(),
13. Description = reader["Description"].ToString(),
14. ProductCategory = new ProductCategoryData
15. {
16. Subcategory = reader["SubcategoryName"].ToString(),
17. Category = reader["CategoryName"].ToString(),
18. },
19. Documentation = new ProductDocumentData
20. {
21. DocumentTitle = reader["Title"].ToString(),
22. DocumentSummary = reader["DocumentSummary"].ToString(),
23. Document = await UploadDataToBlobStorageAsync(reader["Document"], reader["DocumentFileName"].ToString())
24. },
25. Images = new ProductImageData
26. {
27. Diagram = await UploadDataToBlobStorageAsync(reader["Diagram"], reader["ModelName"].ToString()),
28. Thumbnail = await UploadDataToBlobStorageAsync(reader["ThumbnailPhoto"], reader["ThumbnailPhotoFileName"].ToString()),
29. LargePhoto = await UploadDataToBlobStorageAsync(reader["LargePhoto"], reader["LargePhotoFileName"].ToString())
30. }

};

1. Under the comment **TODO: Upload the document to the collection in Cosmos DB**, type the following code:
2. if (await UploadDocumentToCosmosDB(productDocument))
3. {
4. Console.WriteLine($"Added document {productDocument.ProductID} for {productDocument.ProductName}: {productDocument.Description}");
5. }
6. else
7. {
8. Console.WriteLine($"Upload failed for {productDocument.ProductID}: {productDocument.ProductName}");

}

1. Under the comment **TODO: If the data is NULL, return an empty string**, type the following code:
2. if (string.Compare(data.GetType().Name, "DBNull") == 0)
3. {
4. return string.Empty;

}

1. Under the comment **TODO: Otherwise cast the data into a byte array**, type the following code:

var byteData = data as byte[];

1. Under the comment **TODO: If the cast failed, the data probably contains a string, so use GetBytes to convert it into a byte array**, type:
2. if (byteData == null)
3. {
4. byteData = Encoding.ASCII.GetBytes(data.ToString());

}

1. Under the comment **TODO: Get a reference to the blob container and a blob in Azure storage. The blob will hold the data being uploaded**, type the following code:
2. var cloudBlobContainer = this.blobClient.GetContainerReference(this.blobContainer);

var blockBlob = cloudBlobContainer.GetBlockBlobReference(name);

1. Under the comment **TODO: Upload the data to the blob**, type the following code:

await blockBlob.UploadFromByteArrayAsync(byteData, 0, byteData.Length);

1. Under the comment **TODO: Return the URI of the new blob as a string**, type the following code:

return blockBlob.Uri.ToString();

1. Under the comment **TODO: Add the document to the Cosmos DB database**, type the following code:
2. Uri productCollection = UriFactory.CreateDocumentCollectionUri(this.database, this.collection);
3. var response = await this.client.CreateDocumentAsync(productCollection, productDocument);

return (response.StatusCode == System.Net.HttpStatusCode.Created);

1. Press F5 to build and run the application.
2. Verify that it retrieves the data from SQL Server and uploads the documents to Cosmos DB without reporting any errors.
3. In Internet Explorer, on the **20777-sql-<*your name>-<the day*>** blade, click **Data Explorer**.
4. In the **SQL API** pane, expand **SalesData**, expand **ProductCatalog**, and then click **Documents**.
5. Click on one of the documents in the list to view it.
6. Hold Ctrl and left click the value of the **/images/thumbnail** or **/images/largephoto** properties to view the associated images.
7. If the Internet Explorer message box appears, click **Options for this site**, and then click **Always allow**.
8. If the image does not appear in a new tab, repeat steps 30 to 32.
9. Close the No Image Available tab.

**Result**: At the end of this exercise, you should have populated your Cosmos DB database with product catalog information.

**Exercise 3: Querying Product Catalog Information**

**Task 1: Create the Adventure-Works Web Application**

1. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project** dialog box, expand **Visual C#**, click **Web**, and then click **ASP.NET Web Application (.NET Framework)**.
3. In the **Name** box, type **Adventure-Works**.
4. In the **Location** box, type **E:\Labfiles\Lab02**, and then click **OK**.
5. In the **New ASP.NET Web Application - Adventure-Works** dialog box, click **MVC**, and then click **OK**.
6. On the **Tools** menu, point to **NuGet Package Manager**, and then click **Manage NuGet Packages for Solution**.
7. On the **NuGet - Solution** tab, click **Browse**.
8. In the **Search** box, type **Microsoft.Azure.DocumentDB**, and then click **Microsoft.Azure.DocumentDB**.
9. In the right-hand pane, select the **Adventure-Works** check box, and then click **Install**.
10. In the **Preview Changes** dialog box, click **OK**.
11. In the **License Acceptance** dialog box, click **I Accept**.
12. On the **NuGet - Solution** tab, click **Updates**, and then clear the search box.
13. Select the **Select all packages** check box, and then click **Update**.
14. In the **Preview Changes** dialog box, click **OK**.
15. In the **License Acceptance** dialog box, click **I Accept**.
16. If a **Restart Visual Studio to finish the process** message appears, click **Restart**.

**Task 2: Define the Models and ViewModels for the Web Application**

1. In Solution Explorer, right-click **Models**, point to **Add**, and then click **Class**.
2. In the **Add New Item - Adventure-Works** dialog box, in the **Name** box, type **Product.cs**, and then click **Add**.
3. In Product.cs, replace:
4. public class Product
5. {

}

with:

public class ProductCategoryData

{

[JsonProperty("subcategory")]

public string Subcategory { get; set; }

[JsonProperty("category")]

public string Category { get; set; }

}

public class ProductDocumentData

{

[JsonProperty("documenttitle")]

public string DocumentTitle { get; set; }

[JsonProperty("documentsummary")]

public string DocumentSummary { get; set; }

[JsonProperty("document")]

public string Document { get; set; }

}

public class ProductImageData

{

[JsonProperty("diagram")]

public string Diagram { get; set; }

[JsonProperty("thumbnail")]

public string Thumbnail { get; set; }

[JsonProperty("largephoto")]

public string LargePhoto { get; set; }

}

public class Product

{

[JsonProperty("id")]

public string ProductID { get; set; }

[JsonProperty("productname")]

public string ProductName { get; set; }

[JsonProperty("productnumber")]

public string ProductNumber { get; set; }

[JsonProperty("color")]

public string Color { get; set; }

[JsonProperty("listprice")]

public decimal ListPrice { get; set; }

[JsonProperty("size")]

public string Size { get; set; }

[JsonProperty("weight")]

public string Weight { get; set; }

[JsonProperty("quantityinstock")]

public int QuantityInStock { get; set; }

[JsonProperty("model")]

public string Model { get; set; }

[JsonProperty("description")]

public string Description { get; set; }

[JsonProperty("productcategory")]

public ProductCategoryData ProductCategory { get; set; }

[JsonProperty("documentation")]

public ProductDocumentData Documentation { get; set; }

[JsonProperty("images")]

public ProductImageData Images { get; set; }

}

1. At the beginning of the file, type the following code:

using Newtonsoft.Json;

1. In Solution Explorer, right-click **Models**, point to **Add**, and then click **Class**.
2. In the **Add New Item - Adventure-Works** dialog box, in the **Name** box, type **ViewModels.cs**, and then click **Add**.
3. In ViewModels.cs, replace:
4. public class ViewModels
5. {

}

with:

public class ProductCategoryViewModel

{

public string CategoryName { get; set; }

public IEnumerable<SelectListItem> Categories { get; set; }

}

public class ProductSubcategoryViewModel

{

public string SubcategoryName { get; set; }

public IEnumerable<SelectListItem> Subcategories { get; set; }

}

public class ProductViewModel

{

public string SearchString { get; set; }

public IEnumerable<Product> Products { get; set; }

public ProductCategoryViewModel SelectableCategories { get; set; }

public ProductSubcategoryViewModel SelectableSubcategories { get; set; }

}

1. At the beginning of the file, type the following code:

using System.Web.Mvc;

**Task 3: Implement a Repository Class that Connects to the Cosmos DB Database**

1. In Solution Explorer, right-click **Adventure-Works** (not **Solution 'Adventure-Works'**), point to **Add**, and then click **Class**.
2. In the **Add New Item - Adventure-Works** dialog box, in the **Name** box, type **ProductsRepository.cs**, and then click **Add**.
3. In ProductsRepository.cs, replace:
4. public class ProductsRepository
5. {

}

with:

public static class ProductsRepository<T>

{

}

1. At the beginning of the file, type the following code:
2. using Microsoft.Azure.Documents.Client;
3. using Microsoft.Azure.Documents.Linq;
4. using System.Configuration;
5. using System.Linq.Expressions;

using System.Threading.Tasks;

1. Inside the class definition:
2. public static class ProductsRepository<T>

{

type:

private static string endpointUrl;

private static string primaryKey;

private static string database;

private static string collection;

private static DocumentClient client;

1. Type the following code:
2. public static void Initialize(string coll)
3. {
4. endpointUrl = ConfigurationManager.AppSettings["EndpointUrl"];
5. primaryKey = ConfigurationManager.AppSettings["PrimaryKey"];
6. database = ConfigurationManager.AppSettings["Database"];
7. collection = coll;
8. client = new DocumentClient(new Uri(endpointUrl), primaryKey);

}

1. Type the following code:
2. public static async Task<IEnumerable<T>> GetAllItemsAsync()
3. {
4. // Find all documents in the collection
5. IDocumentQuery<T> query = client.CreateDocumentQuery<T>(
6. UriFactory.CreateDocumentCollectionUri(database, collection))
7. .AsDocumentQuery();
8. // Return the documents as a list
9. List<T> results = new List<T>();
10. while (query.HasMoreResults)
11. {
12. results.AddRange(await query.ExecuteNextAsync<T>());
13. }
14. return results;

}

1. Type the following code:
2. public static async Task<IEnumerable<T>> GetItemsAsync(Expression<Func<T, bool>> predicate)
3. {
4. // Find all documents in the collection that match the predicate
5. IDocumentQuery<T> query = client.CreateDocumentQuery<T>(
6. UriFactory.CreateDocumentCollectionUri(database, collection),
7. new FeedOptions { EnableCrossPartitionQuery = true })
8. .Where(predicate)
9. .AsDocumentQuery();
10. // Return the matching documents as a list
11. List<T> results = new List<T>();
12. while (query.HasMoreResults)
13. {
14. results.AddRange(await query.ExecuteNextAsync<T>());
15. }
16. return results;

}

1. In Solution Explorer, double-click **Web.config**.
2. In Web.config, add the following keys to the **appSettings** section:
3. <add key="EndpointUrl" value="~URL~" />
4. <add key="PrimaryKey" value="~PRIMARY KEY~" />
5. <add key="Database" value="SalesData" />
6. <add key="ProductsCollection" value="ProductCatalog" />

<add key="CategoriesCollection" value="ProductCategories" />

The finished section should look like:

<configuration>

<appSettings>

<add key="EndpointUrl" value="~URL~" />

<add key="PrimaryKey" value="~PRIMARY KEY~" />

<add key="Database" value="SalesData" />

<add key="ProductsCollection" value="ProductCatalog" />

<add key="CategoriesCollection" value="ProductCategories" />

<add key="webpages:Version" value="3.0.0.0" />

<add key="webpages:Enabled" value="false" />

...

</appSettings>

...

...

</configuration>

1. In Web.config, in the value of the **EndpointUrl** key, paste the **URI** value you noted earlier, replacing the text **~URL~**.
2. In Web.config, in the value of the **PrimaryKey** key, paste the **PRIMARY KEY** value you noted earlier, replacing the text **~PRIMARY KEY~**.
3. In Solution Explorer, double-click **Global.asax**.
4. In Global.asax.cs, at the end of the **Application\_Start** method, type the following code:
5. ProductsRepository<Adventure\_Works.Models.Product>.Initialize(ConfigurationManager.AppSettings["ProductsCollection"]);

ProductsRepository<Adventure\_Works.Models.ProductCategoryData>.Initialize(ConfigurationManager.AppSettings["CategoriesCollection"]);

The finished method should look like:

protected void Application\_Start()

{

...

ProductsRepository<Adventure\_Works.Models.Product>.Initialize(ConfigurationManager.AppSettings["ProductsCollection"]);

ProductsRepository<Adventure\_Works.Models.ProductCategoryData>.Initialize(ConfigurationManager.AppSettings["CategoriesCollection"]);

}

1. At the beginning of the file, type the following code:

using System.Configuration;

**Task 4: Create the Products Controller and Initial View**

1. In Solution Explorer, right-click **Controllers**, point to **Add**, and then click **Controller**.
2. In the **Add Scaffold** dialog box, click **MVC 5 Controller - Empty**, and then click **Add**.
3. In the **Add Controller** dialog box, in the **Controller name** box, type **ProductsController**, and then click **Add**.
4. In ProductsController.cs, at the beginning of the file, type the following code:
5. using Adventure\_Works.Models;

using System.Threading.Tasks;

1. In the **ProductsController** class, after the **Index** method, type the following code:
2. private async Task<IEnumerable<ProductCategoryData>> InitializeCategoryDataAsync()
3. {
4. // Retrieve the list of categories and subcategories from the document database
5. var categoryData = await ProductsRepository<ProductCategoryData>.GetAllItemsAsync();
6. Session["categoryData"] = categoryData;
7. return categoryData;

}

1. Type the following code:
2. private async Task<ProductCategoryViewModel> InitializeCategoriesAsync()
3. {
4. // Construct the ProductCategoryViewModel data using the data retrieved from the document database
5. var categoryData = Session["categoryData"] as IEnumerable<ProductCategoryData> ?? await ProductsRepository<ProductCategoryData>.GetAllItemsAsync();
6. var distinctCategories = (from c in categoryData select c.Category).Distinct();
7. var selectableCategories = new ProductCategoryViewModel { Categories = from c in distinctCategories select new SelectListItem { Text = c, Value = c } };
8. Session["selectableCategories"] = selectableCategories;
9. return selectableCategories;

}

1. Type the following code:
2. private async Task<ProductSubcategoryViewModel> InitializeSubCategoriesAsync()
3. {
4. // Construct the ProductSubcategoryViewModel data using the data retrieved from the document database
5. var categoryData = Session["categoryData"] as IEnumerable<ProductCategoryData> ?? await ProductsRepository<ProductCategoryData>.GetAllItemsAsync();
6. var subcategoryData = (from c in categoryData select c.Subcategory);
7. var selectableSubcategories = new ProductSubcategoryViewModel { Subcategories = from s in subcategoryData select new SelectListItem { Text = s, Value = s } };
8. Session["selectableSubcategories"] = selectableSubcategories;
9. return selectableSubcategories;

}

1. Type the following code:
2. // Initial, default action for the form
3. [ActionName("FindProducts")]
4. [HttpGet]
5. public async Task<ActionResult> FindProductsAsync()
6. {
7. // Build and display an initial view model containing just the list of categories and subcategories (don't display all products by default as there could be a lot of them)
8. return View(new ProductViewModel
9. {
10. Products = new List<Product>(),
11. SelectableCategories = Session["selectableCategories"] as ProductCategoryViewModel ?? await InitializeCategoriesAsync(),
12. SelectableSubcategories = Session["selectableSubcategories"] as ProductSubcategoryViewModel ?? await InitializeSubCategoriesAsync()
13. });

}

1. Delete the **Index** method from the **ProductsController** class:
2. // GET: Products
3. public ActionResult Index()
4. {
5. return View();

}

1. In Solution Explorer, expand **Views**, right-click **Products**, point to **Add**, and then click **View**.
2. In the **Add View** dialog box, in the **View name** box, type **FindProducts**.
3. In the **Template** drop-down list, click **Empty (without model)**, and then ensure that **Use a layout page** is selected.
4. In the **Use a layout page** box, type **~/Views/Shared/\_Layout.cshtml**, and then click **Add**.
5. In FindProducts.cshtml, at the beginning of the file, type the following code:

@model Adventure\_Works.Models.ProductViewModel

1. Replace:

<h2>FindProducts</h2>

with:

<h1>Product Catalog</h1>

<div class="panel panel-default">

<div class="panel panel-body">

@using (Html.BeginForm("FindProducts", "Products", FormMethod.Post, new { role = "form" }))

{

<div class="form-group">

<h2>Categories</h2>

@Html.DropDownListFor(c => c.SelectableCategories.CategoryName, new SelectList(Model.SelectableCategories.Categories, "Text", "Value"), new { @class = "form-control" })

</div>

<p>

<button type="submit" class="btn btn-primary">Search By Category</button>

</p>

}

@using (Html.BeginForm("FindProducts", "Products", FormMethod.Post, new { role = "form" }))

{

<div class="form-group">

<h2>Subcategories</h2>

@Html.DropDownListFor(s => s.SelectableSubcategories.SubcategoryName, new SelectList(Model.SelectableSubcategories.Subcategories, "Text", "Value"), new { @class = "form-control" })

</div>

<p>

<button type="submit" class="btn btn-primary">Search By Subcategory</button>

</p>

}

@using (Html.BeginForm("FindProducts", "Products", FormMethod.Post, new { role = "form" }))

{

<div class="form-group">

<h2>Product Name, Model Name, or Product Number</h2>

@Html.TextBoxFor(t => t.SearchString, new { @class = "form-control" })

</div>

<p>

<button type="submit" class="btn btn-primary">Search By Product Name, Model Name, or Product Number</button>

</p>

}

<!--TODO: Table showing matching products/-->

</div>

</div>

1. In Solution Explorer, right-click **Adventure-Works** (not **Solution 'Adventure-Works'**), and then click **Properties**.
2. In the left pane, click **Web**.
3. On the **Web** page, under **Start Action**, click **Specific Page**, but leave the page name empty.
4. On the **File** menu, click **Save Selected Items**.
5. In Solution Explorer, expand **App\_Start**, and then double-click **RouteConfig.cs**.
6. In RouteConfig.cs, after the line that begins **routes.IgnoreRoute** on a new line, type the following code:
7. routes.MapRoute(
8. name: "Home",
9. url: "",
10. defaults: new { controller = "Products", action = "FindProducts" }

);

1. In Solution Explorer, expand **Views**, expand **Shared**, and then double-click **\_Layout.cshtml**.
2. In \_Layout.cshtml, edit the contents of so that it reads:
3. <!DOCTYPE html>
4. <html>
5. <head>
6. <meta charset="utf-8" />
7. <meta name="viewport" content="width=device-width, initial-scale=1.0">
8. <title>@ViewBag.Title Adventure-Works Cycles</title>
9. @Styles.Render("~/Content/css")
10. @Scripts.Render("~/bundles/modernizr")
11. </head>
12. <body>
13. <div class="navbar navbar-inverse navbar-fixed-top">
14. <div class="container">
15. <div class="navbar-header">
16. <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
17. <span class="icon-bar"></span>
18. <span class="icon-bar"></span>
19. <span class="icon-bar"></span>
20. </button>
21. @Html.ActionLink("Adventure-Works Product Catalog", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
22. </div>
23. <div class="navbar-collapse collapse">
24. <ul class="nav navbar-nav">
25. <li>@Html.ActionLink("Home", "Index", "Home")</li>
26. <li>@Html.ActionLink("Catalog", "FindProducts", "Products")</li>
27. <li>@Html.ActionLink("About", "About", "Home")</li>
28. <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
29. </ul>
30. </div>
31. </div>
32. </div>
33. <div class="container body-content">
34. @RenderBody()
35. <hr />
36. <footer>
37. <p>&copy; @DateTime.Now.Year - Adventure-Works Cycles</p>
38. </footer>
39. </div>
40. @Scripts.Render("~/bundles/jquery")
41. @Scripts.Render("~/bundles/bootstrap")
42. @RenderSection("scripts", required: false)
43. </body>

</html>

1. Press F5 to build and run the web application.
2. Verify that the **Product Catalog** page appears. The **Categories** and **Subcategories** lists should be populated (click the drop-down lists to see the data). Note that if you click any of the buttons, you will currently receive an error because you have not implemented the code that handles the POST request yet.
3. Close the web application.

**Task 5: Extend the Products Controller to Fetch and Display Product Data**

1. In Visual Studio, in Solution Explorer, double-click **ProductsController.cs**.
2. In ProductsController.cs, after the **FindProductsAsync** method, type the following code:
3. // Method called as POST request from the various search buttons on the form
4. [ActionName("FindProducts")]
5. [HttpPost]
6. public async Task<ActionResult> FindProductsAsync(ProductViewModel productViewModel)
7. {
8. // Construct a view model that matches the search criteria specified by the user
9. IEnumerable<Product> products = null;
10. if (productViewModel.SelectableCategories != null)
11. {
12. // User is searching by category
13. products = await ProductsRepository<Product>.GetItemsAsync(p => p.ProductCategory.Category == productViewModel.SelectableCategories.CategoryName);
14. }
15. else if (productViewModel.SelectableSubcategories != null)
16. {
17. // User is searching by subcategory
18. products = await ProductsRepository<Product>.GetItemsAsync(p => p.ProductCategory.Subcategory == productViewModel.SelectableSubcategories.SubcategoryName);
19. }
20. else
21. {
22. // User is searching by product name, model name, or product number
23. products = await ProductsRepository<Product>.GetItemsAsync(p =>
24. p.ProductName.Contains(productViewModel.SearchString) ||
25. p.Model.Contains(productViewModel.SearchString) ||
26. p.ProductNumber.Contains(productViewModel.SearchString));
27. }
28. // Construct a new view model containing the results and display it
29. return View(new ProductViewModel
30. {
31. Products = products,
32. SelectableCategories = Session["selectableCategories"] as ProductCategoryViewModel ?? await InitializeCategoriesAsync(),
33. SelectableSubcategories = Session["selectableSubcategories"] as ProductSubcategoryViewModel ?? await InitializeSubCategoriesAsync()
34. });

}

1. In Solution Explorer, double-click **FindProducts.cshtml** file.
2. In FindProducts.cshtml, after **<!--TODO: Table showing matching products/-->**, type the following code:
3. <table class="table">
4. <thead>
5. <tr>
6. <td>Product Number</td>
7. <td>Product Name</td>
8. <td>Model</td>
9. <td></td>
10. </tr>
11. </thead>
12. @foreach (var item in Model.Products)
13. {
14. <tr>
15. <td>
16. @Html.DisplayFor(modelItem => item.ProductNumber)
17. </td>
18. <td>
19. @Html.DisplayFor(modelItem => item.ProductName)
20. </td>
21. <td>
22. @Html.DisplayFor(modelItem => item.Model)
23. </td>
24. <td>
25. @Html.ActionLink("Details", "GetProductDetails", new { productID = item.ProductID })
26. </td>
27. </tr>
28. }

</table>

1. Press F5 to build and run the web application.
2. When the application starts, in the **Categories** list, click **Components**, and then click **Search By Category**. The view should now display all products that match that category.
3. In the **Subcategories** list, click **Road Bikes**, and then click **Search By Subcategory**. This time the view should display only those products in that subcategory.
4. In the **Product Name, Model Name, or Product Number** box, type **Wash**, and then click **Search By Product Name, Model Name, or Product Number**. You should now be presented with all products that contain the text **Wash** in the product number, name, or model.
5. If you click the **Details** link for a product, the application will currently report an error because you haven't implemented the **GetProductDetails** method yet.
6. Close the web application and return to Visual Studio.
7. On the **ProductsController.cs** tab, after the second **FindProductsAsync** method (that you added earlier in this exercise), type the following code:
8. // Method called to display the details of the specified product
9. [ActionName("GetProductDetails")]
10. [HttpGet]
11. public async Task<ActionResult> GetProductDetailsAsync(string productID)
12. {
13. // Find the data for the specified product
14. var products = await ProductsRepository<Product>.GetItemsAsync(p => p.ProductID == productID);
15. // Display the details of the product
16. return View(products.First());

}

1. In Solution Explorer, expand **Views**, right-click **Products**, point to **Add**, and then click **View**.
2. In the **Add View** dialog box, in the **View name** box, type **GetProductDetails**.
3. In the **Template** box, click **Empty (without model)**, and then ensure **Use a layout page** is selected.
4. In the **Use a layout page** box, type **~/Views/Shared/\_Layout.cshtml**, and then click **Add**.
5. In GetProductDetails.cshtml, edit the contents of the file so that it reads:
6. @model Adventure\_Works.Models.Product
7. @{
8. ViewBag.Title = "Products";
9. Layout = "~/Views/Shared/\_Layout.cshtml";
10. }

<h1>Product Details</h1>

1. Type the following code:
2. <div class="panel panel-default">
3. <div class="panel panel-body">
4. <div class="form-group">
5. <table class="table">
6. <tr>
7. <td>Product Number: </td>
8. <td>@Html.DisplayTextFor(p => p.ProductNumber)</td>
9. </tr>
10. <tr>
11. <td>Product Name: </td>
12. <td>@Html.DisplayTextFor(p => p.ProductName)</td>
13. </tr>
14. <tr>
15. <td>Model: </td>
16. <td>@Html.DisplayTextFor(p => p.Model)</td>
17. </tr>
18. <tr>
19. <td>Description: </td>
20. <td>@Html.DisplayTextFor(p => p.Description)</td>
21. </tr>
22. <tr>
23. <td>Price: </td>
24. @if (Model.ListPrice == 0)
25. {
26. <td><p>Contact Adventure-Works for price information</p></td>
27. }
28. else
29. {
30. <td>@Html.DisplayTextFor(p => p.ListPrice)</td>
31. }
32. </tr>
33. <tr>
34. <td>Color: </td>
35. <td>@Html.DisplayTextFor(p => p.Color)</td>
36. </tr>
37. <tr>
38. <td>Size: </td>
39. <td>@Html.DisplayTextFor(p => p.Size)</td>
40. </tr>
41. <tr>
42. <td>Weight: </td>
43. <td>@Html.DisplayTextFor(p => p.Weight)</td>
44. </tr>
45. <tr>
46. <td>In Stock: </td>
47. <td>@Html.DisplayTextFor(p => p.QuantityInStock)</td>
48. </tr>
49. @if (Model.Documentation.Document.Length > 0)
50. {
51. <tr>
52. <td>Documents: </td>
53. <td>
54. @Html.DisplayTextFor(p => p.Documentation)
55. <a href="@Model.Documentation.Document">Click to download</a>
56. </td>
57. </tr>
58. }
59. <tr>
60. <td>Image: </td>
61. <td>
62. <img src="@Model.Images.LargePhoto" />
63. </td>
64. </tr>
65. </table>
66. </div>
67. </div>

</div>

1. Press F5 to build and run the web app.
2. When the application starts, in the **Categories** list, click **Components**, and then click **Search By Category**.
3. From the results list, click any of the **Details** links to view the product details.
4. Close the web application.

**Result**: At the end of this exercise, you should have created an application that you can use to query product information.

**Exercise 4: Maintaining Stock Levels in the Product Catalog**

**Task 1: Create a Collection and Repository for Storing Back Orders**

1. In Internet Explorer, in the **SQL API** pane, right-click **SalesData**, and then click **New Collection**.
2. On the **Add Collection** blade, in the **Collection Id** box, type **BackOrders**, and then click **Unlimited**.
3. In the **Partition key** box, type **/productid**.
4. In the **Throughput (400 - 1,000,000 RU/s)** box, type **1000**, and then click **OK**.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, in the **File name** box, type **E:\Labfiles\Lab02\Starter\Adventure-Works-Ex4\Adventure-Works.sln**, and then click **Open**.
7. In the **Security Warning for MigrateProductData** dialog box, clear the **Ask me for every project in this solution** check box, click **OK**.
8. In Solution Explorer, double-click **Web.config**.
9. In Web.config, in the value of the **EndpointUrl** key, paste the **URI** value you noted earlier, replacing the text **~URL~**.
10. In Web.config, in the value of the **PrimaryKey** key, paste the **PRIMARY KEY** value you noted earlier, replacing the text **~PRIMARY KEY~**.
11. In Web.config, add the following setting to the list of app settings:
12. <appSettings>
13. ...
14. <add key="BackOrderCollection" value="BackOrders"/>
15. ...

</appSettings>

1. In Solution Explorer, expand **Models**, and then double-click **Product.cs**.
2. In Product.cs, after the definition of the **Product** class, type the following code:
3. public class ProductBackOrder
4. {
5. [JsonProperty("id")]
6. public string BackOrderID { get; set; }
7. [JsonProperty("productid")]
8. public string ProductID { get; set; }
9. [JsonProperty("productname")]
10. public string ProductName { get; set; }
11. [JsonProperty("productnumber")]
12. public string ProductNumber { get; set; }
13. [JsonProperty("numberonbackorder")]
14. public int NumberOnBackOrder { get; set; }
15. [JsonProperty("backorderdate")]
16. public DateTime BackOrderDate { get; set; }

}

1. In Solution Explorer, double-click **Global.asax**.
2. In global.asax.cs, edit the **Application\_Start** method to add the following line:

ProductsRepository<Adventure\_Works.Models.ProductBackOrder>.Initialize(ConfigurationManager.AppSettings["BackOrderCollection"]);

When you have finished, the **Application\_Start** method should look like this:

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

ProductsRepository<Adventure\_Works.Models.Product>.Initialize(ConfigurationManager.AppSettings["ProductsCollection"]);

ProductsRepository<Adventure\_Works.Models.ProductCategoryData>.Initialize(ConfigurationManager.AppSettings["CategoriesCollection"]);

ProductsRepository<Adventure\_Works.Models.ProductBackOrder>.Initialize(ConfigurationManager.AppSettings["BackOrderCollection"]);

}

**Task 2: Extend the Repository Class to Enable Document Creation and Update**

1. In Solution Explorer, double-click **ProductsRepository.cs**.
2. In ProductsRepository.cs, at the beginning of the file, type the following code:
3. using System.Net;

using Microsoft.Azure.Documents;

1. At the end of the **ProductsRepository<T>** class, type the following code:
2. public static async Task<bool> CreateItem(T item)
3. {
4. // Add the new document to the collection
5. Uri collectionUri = UriFactory.CreateDocumentCollectionUri(database, collection);
6. var response = await client.CreateDocumentAsync(collectionUri, item);
7. // Examine the status of the response, to determine whether the document was created successfully or not
8. return response.StatusCode == HttpStatusCode.Created;

}

1. Type the following code:
2. public static async Task<int> IncrementNumericValueInDocumentAsync(string docID, string property, int value, string partitionKey = null)
3. {
4. // Fetch the doc to be updated
5. Uri docUri = UriFactory.CreateDocumentUri(database, collection, docID);
6. var options = partitionKey == null ? null : new RequestOptions
7. {
8. PartitionKey = new PartitionKey(partitionKey)
9. };
10. var response = await client.ReadDocumentAsync(docUri, options);
11. var doc = response.Resource;
12. // Change the value of the specified property
13. var propVal = doc.GetPropertyValue<int>(property);
14. propVal += value;
15. doc.SetPropertyValue(property, propVal);
16. // Attempt to write the modified document back to the database
17. // If another user modifies the same doc at the same time, this code will throw an exception back to the caller where it should be handled
18. options = new RequestOptions
19. {
20. AccessCondition = new AccessCondition
21. {
22. Condition = doc.ETag,
23. Type = AccessConditionType.IfMatch
24. }
25. };
26. response = await client.ReplaceDocumentAsync(doc.SelfLink, doc, options);
27. // Return the new value of the property that was updated
28. return propVal;

}

**Task 3: Add Buy Functionality to the Product Details View**

1. In Solution Explorer, expand **Views**, expand **Products**, and then double-click **GetProductDetails.cshtml**.
2. In GetProductDetails.cshtml, after the closing **</table>** tag (line 72), type the following code:
3. @using (Html.BeginForm("BuyProduct", "Products", FormMethod.Post, new { role = "form" }))
4. {
5. <div class="form-group">
6. <p>Number Required:
7. <input name="productID" type="hidden" value="@Model.ProductID" />
8. <input name="subcategory" type="hidden" value="@Model.ProductCategory.Subcategory" />
9. <input name="productName" type="hidden" value="@Model.ProductName" />
10. <input name="productNumber" type="hidden" value="@Model.ProductNumber" />
11. <input id="numRequired" name="numRequired" class="form-control" type="number" value="1" min="1"/></p>
12. <p>
13. <button type="submit" class="btn btn-primary">Purchase</button>
14. </p>
15. </div>

}

1. At the end of the file, type the following code:
2. <style>
3. #numRequired {
4. width: 6em;
5. }

</style>

1. In Solution Explorer, expand **Controllers**, and then double-click **ProductsController.cs**.
2. In ProductsController.cs, after the definition of the **GetProductDetailsAsync** class, type the following code:
3. // Method called when the user wishes to purchase a product
4. [ActionName("BuyProduct")]
5. [HttpPost]
6. public async Task<ActionResult> BuyProductAsync(string productID, string subcategory, string productName, string productNumber, int numRequired)
7. {
8. // Use the repository to update the database and return the new stock level
9. int newStockLevel = await ProductsRepository<Product>.IncrementNumericValueInDocumentAsync(productID, "quantityinstock", -numRequired, subcategory);
10. // If the new stock level is now negative, undo the update and place a back order instead (Adventure-Works currently doesn't allow split orders)
11. if (newStockLevel < 0)
12. {
13. await ProductsRepository<Product>.IncrementNumericValueInDocumentAsync(productID, "quantityinstock", numRequired, subcategory);
14. var backOrder = new ProductBackOrder
15. {
16. BackOrderID = Guid.NewGuid().ToString(),
17. ProductID = productID,
18. ProductName = productName,
19. ProductNumber = productNumber,
20. NumberOnBackOrder = numRequired,
21. BackOrderDate = DateTime.UtcNow
22. };
23. await ProductsRepository<ProductBackOrder>.CreateItem(backOrder);
24. }
25. return View(newStockLevel);

}

1. In Solution Explorer, expand **Views**, right-click **Products**, point to **Add**, and then click **View**.
2. In the **Add View** dialog box, in the **View name** box, type **BuyProduct**.
3. In the **Template** box, click **Empty (without model)**, and ensure that **Use a layout page** is selected.
4. In the **Use a layout page** box, type **~/Views/Shared/\_Layout.cshtml**, and then click **Add**.
5. In BuyProduct.cshtml, edit the contents so that it reads:
6. @model int
7. @{
8. ViewBag.Title = "Products";
9. Layout = "~/Views/Shared/\_Layout.cshtml";
10. }
11. <h1>Product Details</h1>
12. <div class="panel panel-default">
13. <div class="panel panel-body">
14. @if (Model < 0)
15. {
16. <p>Backorder created</p>
17. } else
18. {
19. <p>Order placed</p>
20. }
21. </div>

</div>

1. Press F5 to build and run the web app.
2. When the application starts, in the **Categories** list, click **Components**, and then click **Search By Category**.
3. In the results list, click any of the **Details** links to view the product details.
4. Click **Purchase** to purchase an item; you should see the message **Order placed**.
5. Click the browser back button.
6. In the **Number Required** box, type **5000**, and then click **Purchase**. You should see the message **Backorder created**.
7. Close the web application, then close Visual Studio.
8. In Internet Explorer, on the **SQL API** pane, expand **SalesData**, expand **BackOrders**, and then click **Documents**. Observe that a back order document was created.

**Task 4: Lab Close Down**

To reduce your costs delete the resource group containing your Cosmos DB databases:

1. In Internet Explorer, in the Azure Portal, click **Resource groups**.
2. Right-click the **20777\_Mod02** resource group, and then click **Delete resource group**.
3. On the **Are you sure you want to delete "20777\_Mod02"?** blade, in the **Type the resource group name:** box, type **20777\_Mod02**, and then click **Delete**.
4. When you have finished, close Internet Explorer.

**Result**: In this lab, you have used what you have learned in this lesson to design a document schema, load data into a collection, and to query, modify, and delete documents in a collection.

**Module 3: Implementing Server-Side Operations**

* [Module 3: Implementing Server-Side Operations](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#module-3-implementing-server-side-operations)
  + [Lab: Writing User-Defined Functions, Stored Procedures and Triggers](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#lab-writing-user-defined-functions-stored-procedures-and-triggers)
    - [Lab Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#lab-scenario)
    - [Objectives](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#objectives)
    - [Lab Setup](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#lab-setup)
  + [Exercise 1: Redesigning the Document Structure and Partitioning Strategy](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#exercise-1-redesigning-the-document-structure-and-partitioning-strategy)
    - [Exercise 1 Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#exercise-1-scenario)
    - [Task 1: Prepare the Environment](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-1-prepare-the-environment)
    - [Task 2: Analyze the Technical Requirements for Document Partitioning](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-2-analyze-the-technical-requirements-for-document-partitioning)
    - [Task 3: Implement the Different Types of Documents](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-3-implement-the-different-types-of-documents)
    - [Task 4: Create a Document Database Using the Cosmos DB SQL API](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-4-create-a-document-database-using-the-cosmos-db-sql-api)
    - [Task 5: Import the Product and Customer Data](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-5-import-the-product-and-customer-data)
  + [Exercise 2: Implementing the Shopping Cart Functionality](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#exercise-2-implementing-the-shopping-cart-functionality)
    - [Exercise 2 Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#exercise-2-scenario)
    - [Task 1: Configure the Repository Classes](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-1-configure-the-repository-classes)
    - [Task 2: Complete the Customer Controller to Enable Customers to Login and Logout](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-2-complete-the-customer-controller-to-enable-customers-to-login-and-logout)
    - [Task 3: Implement Functionality to Add Items to the Shopping Cart](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-3-implement-functionality-to-add-items-to-the-shopping-cart)
  + [Exercise 3: Creating Orders and Backorders](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#exercise-3-creating-orders-and-backorders)
    - [Exercise 3 Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#exercise-3-scenario)
    - [Task 1: Create the CheckStockAvailable Stored Procedure](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-1-create-the-checkstockavailable-stored-procedure)
    - [Task 2: Test the CheckStockAvailable Stored Procedure](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-2-test-the-checkstockavailable-stored-procedure)
    - [Task 3: Add Stored Procedure Functionality to the Repository](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-3-add-stored-procedure-functionality-to-the-repository)
    - [Task 4: Call the CheckStockAvailable Stored Procedure From the App Using the Repository](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-4-call-the-checkstockavailable-stored-procedure-from-the-app-using-the-repository)
  + [Exercise 4: Maintaining Data and Auditing Orders and Backorders](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#exercise-4-maintaining-data-and-auditing-orders-and-backorders)
    - [Exercise 4 Scenario](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#exercise-4-scenario)
    - [Task 1: Create a Pre-Trigger that Completes the Details of an Order](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-1-create-a-pre-trigger-that-completes-the-details-of-an-order)
    - [Task 2: Create a Post-trigger that Audits Changes Made to Orders and Backorders](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-2-create-a-post-trigger-that-audits-changes-made-to-orders-and-backorders)
    - [Task 3: Add Trigger Functionality to the Repository](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-3-add-trigger-functionality-to-the-repository)
    - [Task 4: Invoke the Triggers](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-4-invoke-the-triggers)
    - [Task 5: Lab close down](https://github.com/MicrosoftLearning/20777A---Implementing-Microsoft-Azure-Cosmos-DB-Solutions/blob/master/Instructions/20777A_LAB_03.md#task-5-lab-close-down)

**Lab: Writing User-Defined Functions, Stored Procedures and Triggers**

**Lab Scenario**

You have been asked to design an online retail system for a global organization that has thousands of product lines and handles requests from tens of thousands of customers simultaneously. You will be implementing the processing for creating and managing orders. The application should provide the following functionality:

* Customers can browse the Adventure-Works web site and place items in a virtual shopping cart.
* Customers can add or remove items from the shopping cart, and change the number of each item ordered.
* When the customer is ready, the customer can place the order. The application should establish—for each item in the shopping cart—whether there is sufficient stock available and if necessary create a backorder for an item.
* In the case of insufficient availability, the order will be split. The items available will be ordered whilst the remainder will be placed on backorder.
* A customer can browse their orders and backorders, and if the items have not been shipped or delivered, the customer can cancel any of their orders or backorders.
* Whenever an order or backorder is cancelled, an audit document is specifying the order or backorder that was cancelled, and when.
* Customers must not be able to view or amend the shopping carts, orders, or backorders for other customers. Some form of customer authentication and login will be required.

**Objectives**

At the end of this lab, you should be able to:

1. Design and implement a partition scheme suitable for use with server-side programming.
2. Implement an application compatible with your design.
3. Use stored procedures in your applications.
4. Use triggers in your applications..

**Lab Setup**

* **Estimated time**: 120 minutes
* **Virtual machine**: 20777A-LON-DEV
* **User name**: LON-DEV\Administrator
* **Password**: Pa55w.rd

**Exercise 1: Redesigning the Document Structure and Partitioning Strategy**

**Exercise 1 Scenario**

In the prototype version of the Adventure-Works app (that you created in the lab for the previous module), different collections were used to hold different types of data; you stored products in one collection and product category data in another. Once you start adding further types of documents, this approach might not be feasible (or desirable) in a solution, especially if that solution requires stored procedures and triggers to implement server-side processing.

The main tasks for this exercise are as follows:

1. Prepare the environment.
2. Analyze the technical requirements for partitioning.
3. Implement the document types.
4. Create a document database using the Cosmos DB SQL API.
5. Upload data to Cosmos DB.

**Task 1: Prepare the Environment**

Using File Explorer:

* In **E:\Labfiles\Lab03\Starter**, run **Setup.cmd** as Administrator.

Using the Azure portal:

1. Create a new Storage account. Add the account to the **20777\_Mod03** resource group; create this resource group if it does not exist.
2. Create a new blob container. Grant anonymous read access to blobs in the container.
3. Make a note of the **Connection string** for the storage account.

Using the Azure portal:

1. Create a new Cosmos DB account using the SQL API. Add the account to the **20777\_Mod03** resource group.
2. Create a new database named **Adventure-Works**.
3. Make a note of the **URI** and **Primary Key** of the Cosmos DB account.

**Task 2: Analyze the Technical Requirements for Document Partitioning**

Consider the following points:

* Using multiple collections can be more expensive than using a single collection that is partitioned appropriately (because you incur a financial charge for each collection).
* The details of each customer must be stored in the database, to enable them to be identified and authenticated.
* The app should persist customers' shopping carts between sessions. This will require saving them to the database. However, shopping carts should not be retained indefinitely, and after 7 days of inactivity a shopping cart should be removed.
* For reasons of query efficiency, customers, their shopping carts, and their orders should be in the same collection, and ideally in the same partition within that collection. If this collection is partitioned by **customerid**, then performing a query that specifies the **customerid** will automatically retrieve the details of the customer and all of their orders as a single request.
* The Adventure-Works app needs to check stock availability of an item when the customer places an order, and if necessary create one or more backorders; this needs to be an atomic operation, to prevent the same items being allocated to multiple customers who happen to be placing orders concurrently. This suggests that this functionality should be implemented as a stored procedure or trigger.
* Similarly, cancelling an order or cancelling a backorder, and creating the corresponding audit document should also be atomic operations.
* Stored procedures and triggers are scoped to a partition. You specify the partition in which to run a stored procedure when it is invoked. Triggers are automatically scoped to the partition containing the document being inserted, updated, or deleted. In both cases, stored procedures and triggers cannot query, create, modify, or delete data in any other partition.
* Inventory data (the number of items in stock) is stored with each product, and product documents are partitioned by subcategory. If the backorders are stored in the same partition as the order, a stored procedure or trigger will not be able to query product availability and at the same time create new backorders (products and backorders will be in different partitions). This suggests that the backorders for a product must be held in the same partition as the products to which they pertain.
* For the same reason, the audit record of a backorder must be stored in the same partition as the backorder, and the audit record of an order must be stored in the same partition as the order.

**Question**: Assuming that you decide to store the data in a single collection, how should you partition the data? What should you use as the partition key?

**Answer**: Define a field named **partitionkey** for each document. For customers, shopping carts, orders, and order audit documents, this field would represent the customer ID. For products, product category data, backorders, and backorder audit documents, this field would represent the product subcategory—for the reasons covered the lab in the previous module.

**Question**: How should you distinguish the different types of documents in the collection?

**Answer**:Add a field named **doctype** to each document that contains the name of the type of document (**Customer**, **Product**, etc).

**Question**: What is the logical difference between a shopping cart and an order?

**Answer**: Shopping carts and orders are almost the same thing. They both contain a list of order items representing the products being picked or ordered. The primary difference is lifetime. Shopping carts are transitory with a limited duration whereas orders are permanent (they can be cancelled, but the order document should be retained by the system). You can consider the data in a shopping cart to be provisional, and features such the cost of the items are only confirmed when the order is confirmed and any discounts etc are applied.

**Task 3: Implement the Different Types of Documents**

1. Using Visual Studio open the the **Adventure-Works** solution in **E:\Labfiles\Lab03\Starter\Exercise01\Adventure-Works**.
2. Edit the **Web.config** file. Set the **EndpointUrl** and **PrimaryKey** settings to the values you recorded for the Cosmos DB account that you created at the beginning of the lab.
3. In the **Models** folder, edit the **DocumentType.cs** file. This file defines the **DocumentType** class, which will act as an abstract base class for all the different types of documents used by the application. You will implement the functionality that is common to all documents in this class; specifically, the **ttl** and **doctype** fields.
4. After the comment **TODO: doctype distinguishes the different types of documents in the same collection**, add the following code that implements the **doctype** property:
5. [JsonProperty("doctype")]

public string DocType { get; set; }

1. In the **DocumentType** constructor, after the comment **TODO: Initialize the doctype property**, add the following statement. This code stores the type name of the class in the **doctype** field. At runtime, the class name will be the name of the derived class (*Customer*, *ShoppingCartOrder*, etc) that inherits from the **DocumentType** class.

this.DocType = this.GetType().Name;

1. After the comment **TODO: ttl of document. By default, ttl is disabled, but can be changed by setting it to a positive integer value (number of seconds)**, add the following field definition. If a document contains a field named **ttl**, Cosmos DB will use this field as the *time to live* for the document:
2. [JsonProperty("ttl")]

public int TimeToLive { get; set; } = -1;

1. Open the **Customer.cs** file in the **Models** folder. This file contains two classes; **CustomerName** and **Customer**. You will populate these classes with fields that define the details of a customer. Note that **Customer** inherits from **DocumentType**, so it will automatically have a **doctype** field and a **ttl** field.
2. Add the following fields to the **Customer** class. The **name** field contains a subdocument defined using the **CustomerName** class (which is why **CustomerName** is not a descendent of **DocumentType**). Also note that the **partitionkey** field represents the customer ID in the application. The **discountrate** field indicates how much discount the customer will be given on any orders placed:
3. [JsonProperty("partitionkey")]
4. public string CustomerID { get; set; }
6. [JsonProperty("name")]
7. public CustomerName CustomerName { get; set; }
9. [JsonProperty("discountrate")]

public int DiscountRate { get; set; }

1. Add the following fields to the **CustomerName** class:
2. [JsonProperty("title")]
3. public string Title { get; set; }
5. [JsonProperty("firstname")]
6. public string FirstName { get; set; }
8. [JsonProperty("middlename")]
9. public string MiddleName { get; set; }
11. [JsonProperty("lastname")]

public string LastName { get; set; }

1. Open the **ShoppingCartOrder.cs** file in the **Models** folder. This file contains two classes; **OrderItem** and **ShoppingCartOrder**. The same class is used to represent shopping carts and orders. A shopping cart or order will contain one or more order items. Note that **ShoppingCartOrder** inherits from **DocumentType** so it will automatically have a **doctype** field and a **ttl** field.
2. Add the following fields to the **OrderItem** class. When a customer adds an item to the shopping cart, one of these objects will be created. The **NumberOnBackorder**, **BackorderReference**, **unitcost**, and **LineItemTotalCost** fields will only be populated when the shopping cart is converted into an order (prices could change while an item is in the shopping cart; remember that the cart can be retained for up to 7 days).
3. [JsonProperty("productnumber")]
4. public string ProductNumber { get; set; }
6. [JsonProperty("productname")]
7. public string ProductName { get; set; }
9. [JsonProperty("productid")]
10. public string ProductID { get; set; }
12. [JsonProperty("subcategory")]
13. public string Subcategory { get; set; }
15. [JsonProperty("numberincartorordered")]
16. public int NumberInCartOrOrdered { get; set; }
18. [JsonProperty("numberonbackorder")]
19. public int NumberOnBackorder { get; set; }
21. [JsonProperty("backorderreference")]
22. public string BackorderReference { get; set; }
24. [JsonProperty("unitcost")]
25. public decimal UnitCost { get; set; }
27. [JsonProperty("lineitemtotalcost")]

public decimal LineItemTotalCost { get; set; }

1. Add the following fields to the **ShoppingCartOrder** class. The field **IsShoppingCartOrOrder** indicates whether this is a shopping cart or order. As with order items, many of these fields are only filled in once the shopping cart becomes an order. Notice that the **partitionkey** field again represents the customer ID. This ensures that the shopping cart and all orders for a customer are stored in the same partition as the customer.
2. [JsonProperty("id")]
3. public string ShoppingCartOrderID { get; set; }
5. [JsonProperty("partitionkey")]
6. public string CustomerID { get; set; }
8. [JsonProperty("isshoppingcartororder")]
9. public string IsShoppingCartOrOrder { get; set; }
11. [JsonProperty("orderitems")]
12. public List<OrderItem> OrderItems { get; set; }
14. [JsonProperty("numberofitems")]
15. public int NumberOfItems { get; set; }
17. [JsonProperty("itemscost")]
18. public decimal ItemsCost { get; set; }
20. [JsonProperty("customerdiscountrate")]
21. public int CustomerDiscountRate { get; set; } // Indicates % discount
23. [JsonProperty("totalcost")]
24. public decimal TotalCost { get; set; }
26. [JsonProperty("dateplaced")]
27. public long DatePlaced { get; set; } // Stored as ticks
29. [JsonProperty("orderstatus")]
30. public string OrderStatus { get; set; } // "In progress", "Delivered", "Cancelled"
32. [JsonProperty("lastupdated")]

public long LastUpdated { get; set; } // Also stored as ticks

**Task 4: Create a Document Database Using the Cosmos DB SQL API**

1. Add a new collection named **Data** to the **Adventure-Works** database. Use unlimited storage, and **1000** RU/s throughput. Use **/partitionkey** as the partitioning key.
2. Build and run the application. Verify that runs without reporting any errors.

**Task 5: Import the Product and Customer Data**

The application used in this task imports product data from a local SQL Server database file to your Cosmos DB database, and uploads product images and other binary data to Blob storage. It is similar to that used in previous labs, but is updated to reflect the changes in document structure made in this exercise. As in the lab in the previous module, the source data is provided in a SQL Server database file; the application uses the **(localdb)\.** SQL Server instance installed with Visual Studio to open the database file.

1. Using Visual Studio open the the **MigrateProductData** solution in **E:\Labfiles\Lab03\Starter\Exercise01\MigrateProductData**.
2. Edit the **App.config** file. Set the **EndpointUrl** and **PrimaryKey** settings to the values you recorded for the Cosmos DB account that you created at the beginning of the lab. Set the **BlobStorageConnectionString** and **BlobContainer**settings to the values for your storage account that you created at the beginning of the lab.
3. Build and run the application. Verify that it retrieves the data from SQL Server and uploads the documents to Cosmos DB without reporting any errors. Close the Visual Studio instance with the **MigrateProductData** solution open when the upload process is complete.
4. Using the Azure portal, go to Data Explorer for the **Data** collection and examine some of the documents.
5. Use the **Upload** tool in Data Explorer to upload the contents of **E:\Labfiles\Lab03\Data\categories.json** to the **Data**collection. 38 documents should be uploaded.
6. Use the **Upload** tool in Data Explorer to upload the contents of **E:\Labfiles\Lab03\Data\customers.json** to the **Data**collection. 100 documents should be uploaded.

**Result**: At the end of this exercise, you should have revised the partitioning design for the sales application, implemented the new document types, and uploaded data to Cosmos DB.

**Exercise 2: Implementing the Shopping Cart Functionality**

**Note**: In this exercise, you will extend the functionality of the Adventure-Works web application. You can either continue using the solution from Exercise 1, or open the **Adventure-Works** solution in the **E:\Labs\Lab03\Starter\Exercise02\Adventure-Works** folder. These instructions assume that you use the solution in the **E:\Labs\Lab03\Starter\Exercise02\Adventure-Works** folder.

**Exercise 2 Scenario**

The main tasks for this exercise are as follows:

1. Configure the Adventure-Works application repository classes to allow documents to be saved to Cosmos DB.
2. Add a simple customer login and logout process.
3. Implement and test the shopping cart.

**Task 1: Configure the Repository Classes**

1. Using Visual Studio open the the **Adventure-Works.sln** solution in **E:\Labfiles\Lab03\Starter\Exercise02\Adventure-Works**.
2. Edit the **App.config** file. Set the **EndpointUrl** and **PrimaryKey** settings to the values you recorded for the Cosmos DB account that you created at the beginning of the lab. Confirm that the values of the **Database** and **Collection** settings match your Cosmos DB database and collection.

**Note**: Remember that all documents now reside in a single collection.

1. Edit the **Global.asax** file
2. After the comment **TODO: Add Customer repository**, add the following statement. This code creates an instance of the **Repository** class for querying and maintaining **Customer** documents in the Cosmos DB database:

Repository<Customer>.Initialize(ConfigurationManager.AppSettings["Collection"]);

1. After the comment **TODO: Add ShoppingCartOrder repository**, add another statement that creates an instance of the **Repository** class for querying and maintaining **ShoppingCartOrder** documents in the Cosmos DB database:

Repository<ShoppingCartOrder>.Initialize(ConfigurationManager.AppSettings["Collection"]);

**Task 2: Complete the Customer Controller to Enable Customers to Login and Logout**

1. In Solution Explorer, expand the **Controllers** folder, and edit the **CustomerController.cs** file. This file is a controller for managing customers in the app. It contains two methods; one for authenticating customers and logging them in, and another for logging them out.

**Note** These methods are intended to provide simulated security for this app only, and do not reflect best practice for implementing authentication.

1. In the **LoginAsync** method, after the comment **TODO: Use the repository to retrieve the details of the specified customer**, add the following statement:

var customer = (await Repository<Customer>.GetItemsAsync(c => c.CustomerID == customerID)).FirstOrDefault();

1. After the comment **TODO: Validation and authentication using the password etc should go here. Only validation is implemented in this example, the user can enter anything in the password field!** add the following code. If no matching customer is found in the database, the **Login** view (found in the **Shared** views folder in the app) will display the message **Invalid username or password**:
2. if (customer == null) // No such customer
3. {
4. TempData["invalidUserOrPassword"] = "Invalid username or password";
5. return View();

}

1. After the comment **TODO: If authentication is successful (it always is in this example), cache the customer details in the session**, add the following statement. The customer details are held in the Session cache so that they can be accessed from other actions in the various controllers in the app:
2. Session["customer"] = customer;
3. After the comment **TODO: Retrieve the current shopping cart for the customer (if there is one)**, add the following statements. This code fetches the shopping cart for the customer and then caches it in the session. Notice that the query checks the **IsShoppingCartOrOrder** field of the document. A shopping cart has the value **ShoppingCart** in this field as opposed to **Order**. The **Constants** class contains the global constants and strings used throughout the web app.
4. var shoppingCart = (await Repository<ShoppingCartOrder>.GetItemsAsync(c => c.CustomerID == customerID && c.IsShoppingCartOrOrder == Constants.SHOPPINGCART)).FirstOrDefault();

Session["shoppingCart"] = shoppingCart;

1. Review the **Logout** method. All this method does is to remove the details of the customer and other cached items from the session. The **Loggedout** view (another shared view) simply displays a message informing the customer that they have been logged out.

**Task 3: Implement Functionality to Add Items to the Shopping Cart**

1. In Solution Explorer, open the **ShoppingCartController.cs** file in the **Controllers** folder. This file the code for the actions that will run when the customer adds items to the shopping cart, or edits the shopping cart.
2. Find the **ViewCart** method near the start of the **ShoppingCartController** class. This action runs when the customer wishes to view the contents of their shopping cart.
3. After the comment **TODO: Check whether the user has logged in**, add the following code. This block examines the Session cache for the **Customer** object. If it isn't present, the customer is not currently logged in, so the code switches to the shared **Login** view that prompts the customer for their name and password before invoking the **Login** action, that runs the **LoginAsync** method that you completed in the previous task. One the customer has logged in, the Login action uses the information in the cached **ReturnData** object to return to this **ViewCart** action, when the user should now be logged in:
4. if (Session["customer"] == null)
5. {
6. // If not, then get them to log in
7. Session["ReturnData"] = new { ReturnURL = "ViewCart", Controller = "ShoppingCart" };
8. return View("Login");

}

1. After the comment **TODO: Retrieve the shopping cart for the customer**, add the following statement. Remember that when the customer logs in, their shopping cart is cached in the Session. This code simply retrieves the shopping cart from the Session cache:

var shoppingCart = GetShoppingCartForCustomer(Session["customer"] as Customer);

1. After the comment **TODO: Display the shopping cart**, change the **return** statement to pass the shopping cart to the **ViewCart** view for display, as follows:

return View(shoppingCart);

1. In Solution Explorer, in the **Views** folder, expand **ShoppingCart** and review the contents of the **ViewCart.cshtml** file. This is the Razor file that generates the display for the shopping cart. The markup in this file iterates through the **OrderItems** collection in the shopping cart, and displays each line item in turn. Additionally, it displays an edit button for each item that invokes the **EditShoppingCartItem** action that enables the customer to modify the line item, and a delete button that calls the **DeleteShoppingCartItem** action to remove the item from the shopping cart. After all the line items are displayed, a **place order** button appears. If the user clicks this button, the **PlaceOrder** action will run to convert the shopping cart into an order.
2. Open the **GetProductDetails.cshtml** file in the **View/Products** folder in Solution Explorer. Recall from the previous lab that this view is displayed when the customer displays the details of a product when browsing the catalog. In the previous version of the app, the customer could select an item and click **Buy**. This functionality has been replaced with an **Add to Shopping Cart** button which invokes the **AddItemToShoppingCart** action in the **ShoppingCart** controller.
3. Return to the **ShoppingCartController.cs** file, and locate the **AddItemToShoppingCartAsync** method. This method runs when the **AddItemToShoppingCart** action is invoked. The parameter is the number required of the specified product. The code in this method ascertains whether the customer is currently logged in, and if not displays the **Login**view. The code then retrieves the shopping cart for the customer, and fetches the details of the selected product (the product id is previously cached in the session). The method then either adds the product to the shopping cart as a new order item, or if the product is already in the shopping cart the code increments the number required by the appropriate amount.
4. Find the comment **TODO: Save the shopping cart using the repository, and then cache it in the session. Set the TTL to 7 days. Note that TTL must be enabled at the collection level for this to work**. After this comment, add the following code:
5. if (await Repository<ShoppingCartOrder>.UpsertItemAsync(shoppingCart, ttl: 60 \* 60 \* 24 \* 7))
6. {
7. Session["shoppingCart"] = shoppingCart;
9. // Inform the user that the item has been added to the cart
10. TempData["itemMessage"] = "Item added to shopping cart";
11. }
12. else
13. {
14. // If the upsert failed, then inform the user that the item has not been added to the shopping cart
15. TempData["itemMessage"] = "Item not added to shopping cart";

}

1. In the Azure Portal, in Document Explorer, select the **Data** collection and then click **Scale & Settings**. In the **Settings**section of the blade, change **Time to Live** to **On (no default)**, and then click **Save**. This setting causes Cosmos DB to automatically expire documents in the collection once they have exceeded the number of seconds specified in the **ttl**field.
2. Return to Visual Studio. Build and run the application.
3. In the application, browse the catalog and select a product. On the **Product Details** page click **Add to Shopping Cart**. You should be prompted to log in. You can specify any customer id between 1 and 100 (the password is ignored). The message **Item added to shopping cart** should appear, and the contents of the shopping cart should be displayed by the **ViewCart** view.
4. In the drop-down menu, click the **Catalog** link, browse again, and add another item to the cart. When the **ViewCart**page appears again, it should now contain two items. You can use the **Update** button to change the quantity, and the **Delete** button to remove an item. Note that if you click **Place Order**, the application will stop with an error because you haven't yet implemented this feature.
5. Switch back to Document Explorer in the Azure portal. Click **New SQL Query**, and run the following query:

SELECT \* FROM c WHERE c.doctype = "ShoppingCartOrder"

1. Verify that a document similar to that shown below appears. Notice that the **doctype** field is set to **ShoppingCartOrder**, and **IsShoppingCartOrOrder** is set to **ShoppingCart**. The **OrderItems** array contains the items that you added to the cart. You should observe that many of the fields, such as **unitcost**, **totalcost**, and **dateplaced** are 0 or empty. These fields will be givne values when the shopping cart becomes a confirmed order—you will add this feature in the next exercise:
2. [
3. {
4. "id": "3b572667-f49f-4017-ab5c-c137e095fb2f",
5. "partitionkey": "2",
6. "isshoppingcartororder": "ShoppingCart",
7. "orderitems": [
8. {
9. "productnumber": "BK-R93R-56",
10. "productname": "Road-150 Red, 56",
11. "productid": "753",
12. "subcategory": "Road Bikes",
13. "numberincartorordered": 5,
14. "numberonbackorder": 0,
15. "backorderreference": null,
16. "unitcost": 0,
17. "lineitemtotalcost": 0
18. },
19. {
20. "productnumber": "BK-M82B-42",
21. "productname": "Mountain-100 Black, 42",
22. "productid": "776",
23. "subcategory": "Mountain Bikes",
24. "numberincartorordered": 4,
25. "numberonbackorder": 0,
26. "backorderreference": null,
27. "unitcost": 0,
28. "lineitemtotalcost": 0
29. }
30. ],
31. "numberofitems": 0,
32. "itemscost": 0,
33. "customerdiscountrate": 0,
34. "totalcost": 0,
35. "dateplaced": 0,
36. "orderstatus": null,
37. "lastupdated": 0,
38. "doctype": "ShoppingCartOrder",
39. "ttl": -1,
40. "\_rid": "CE1MAMJEDABQAwAAAAAAAA==",
41. "\_self": "dbs/CE1MAA==/colls/CE1MAMJEDAA=/docs/CE1MAMJEDABQAwAAAAAAAA==/",
42. "\_etag": "\"0000e205-0000-0000-0000-5b44e2800000\"",
43. "\_attachments": "attachments/",
44. "\_ts": 1531241088
45. }

]

**Result**: At the end of this exercise, you should have implemented the shopping cart.

**Exercise 3: Creating Orders and Backorders**

**Exercise 3 Scenario**

When a customer places an order, the application must determine whether there is sufficient of each item in stock, allocate stock to the order, and if necessary create one or more backorders. This process should be atomic for each order item. You can implement atomic transactions using triggers, however, in this case a stored procedure is more appropriate. This is because a trigger could only access data in the same partition as the shopping cart, and product details are held in different partitions. The solution is to iterate through the items in the shopping cart and execute a stored procedure for each item. The stored procedure can take the product ID and number required as parameters, together with any other pertinent information. The stored procedure can be run the the scope of the partition in which the product is stored, change the inventory level of the product, and create a backorder in the same partition, if necessary.

The main tasks for this exercise are as follows:

1. Create a stored procedure to check stock levels.
2. Test the new stored procedure.
3. Add the capability to call stored procedures to the application.
4. Update the application to use the new stored procedure to update stock levels.
5. Test the new application functionality.

**Task 1: Create the CheckStockAvailable Stored Procedure**

1. Using the Azure Portal, add a new stored procedure to the **Data** collection.
2. Set the **Id** of the procedure to **CheckStockAvailable**.
3. Start the stored procedure definition with the following code. This code starts the definition of the **CheckStockAvailable** function, that will runs when the stored procedure is called. The parameters are *product ID* and *number required*, as described in the exercise scenario, together with the *product name* and *customer ID*; if it is necessary to create a backorder document, these details will be required. The **Collection** and **collectionLink** variables reference the collection containing the stored procedure. You pass back information from a stored procedure using an HTTP response message:
4. function CheckStockAvailable(productID, productname, numRequired, customerID){
5. var collection = getContext().getCollection();
6. var collectionLink = collection.getSelfLink();

var response = getContext().getResponse();

1. Add the following code to the stored procedure. This code uses the **queryDocuments** function to retrieve the details of the product. It runs a function (an anonymous one in this case) when the query is complete. This function takes three parameters; an object indicating whether an error occurred, an array of documents identified by the query, and an object containing the request options used to run the query. You will complete this function in the following steps:

var isAccepted = collection.queryDocuments(collectionLink, "SELECT \* FROM p WHERE p.doctype = 'Product' AND p.id = '" + productID + "'", {}, function(err, docs, options) {

1. Add the following statement to the anonymous function. This code examines the error object, and if it is not null it rethrows the error. This will cause the stored procedure to abort, and any work performed will be undone. The error should be handled by the client running the stored procedure (the **Adventure-Works** application in this case). You will see how to do this later in the exercise:
2. // If an error has occurred, abort the stored procedure
3. if (err) {
4. throw err;

}

1. Add the following statements to the stored procedure. This code verifies that the query found at least one product. If not, it throws a **not available** error back to the caller:
2. // Check that a doc was found. If not, throw a "not available" error
3. if (!docs || !docs.length) {
4. throw new Error("Product " + productname + " is not available");

}

1. Append the following code to the function. This code checks that there is sufficient quantity of the product in stock. If not, it creates a backorder for the difference between quantity ordered and the quantity available. The **updateStockLevel** function (which you will write later) changes the available stock level of the product in the database. This code then calls the **createDocument** API function to add the backorder document to the database, using the same idiom as before to handle any errors. If the create is successful, the code creates a response message containing a JSON object specifies that a backorder was created, how many of the product was successfully allocated (the backorder covers the remainder), and the latest unit price per item:
2. else {
3. // If a matching product was found, check for sufficient stock levels
4. var productDoc = docs[0];
5. if (numRequired > productDoc.quantityinstock) {
6. // If there is insufficient stock available, allocate what is there
7. var numOnBackorder = numRequired - productDoc.quantityinstock;
8. updateStockLevel(productDoc, productDoc.quantityinstock);
9. // Create a backorder for the remainder
10. var backorder = {
11. doctype: "ProductBackorder",
12. partitionkey: productDoc.partitionkey,
13. customerid: customerID,
14. productid: productDoc.id,
15. productname: productDoc.productname,
16. productnumber: productDoc.productnumber,
17. numberonbackorder: numOnBackorder,
18. backorderstatus: "In progress",
19. backorderdate: getCurrentTimeInNetTicks()
20. };
21. // Add the backorder to the database.
22. // If an error occurs, the stored procedure will be aborted and changes rolled back
23. isAccepted = collection.createDocument(collectionLink, backorder, function(err, backorderDoc, options){
24. // If an error has occurred, abort the stored procedure
25. if (err) {
26. throw err;
27. }
28. // Set the response body to indicate how many items were allocated, and the details of any backorder
29. response.setBody('{message: "backorder created", backorderid: "' + backorderDoc.id + '", allocated: ' + (numRequired - numOnBackorder) + ', latestprice:' + productDoc.listprice + '}');
30. });
31. if (!isAccepted) {
32. throw new Error('Unable to create backorder. Please retry.');
33. }

}

1. Add the following JavaScript statements to the stored procedure. This code handles the case where there is sufficient stock available, so the stock level can be updated without creating a backorder:
2. else {
3. // If there is sufficient stock available, then allocate it
4. updateStockLevel(productDoc, numRequired);
5. response.setBody('{message: "items in stock", allocated: ' + numRequired + ', latestprice:' + productDoc.listprice + '}');
6. }
7. }

});

1. Add the following statements to the code. The previous processing was initiated by calling the **queryDocument**function. However, if the Cosmos DB runtime determines that there is insufficient time available to perform the query, the return value from the **queryDocument** function is false. In this case, the code below causes the stored procedure to be aborted:
2. if (!isAccepted) {
3. throw new Error('Unable to check stock level. Please retry.');

}

1. Add the following function to the end of the JavaScript code. This function uses the JavaScript API to safely update the stock level for a product. It uses the ETag of the product (retrieved previously) to ensure that no other customer has changed the stock level in the meantime:

**Note**: Ideally, rather than throwing an error if an update conflict occurs, this function would fetch the latest version of the product again, test the stock level, create a backorder if necessary, and then modify the stock level accordingly. This process is left as an exercise that you might like to try later, if you have time.

function updateStockLevel(productDoc, numRequired) {

// Update the stock level

productDoc.quantityinstock -= numRequired;

// Save the product doc back to the collection, and check for concurrent updates

// If an error occurs, the stored procedure will be aborted and changes rolled back

isAccepted = collection.replaceDocument(productDoc.\_self, productDoc, {etag: productDoc.Etag});

if (!isAccepted) {

throw new Error('Unable to update stock level. Please retry.');

}

}

1. Finally, add the **getCurrentTimeInNetTicks** function shown below to the code. This function returns the current time in .NET ticks (as opposed to JavaScript ticks):
2. function getCurrentTimeInNetTicks() {
3. // Get the current time
4. var now = new Date().getTime();
5. // Adjust for 1st Jan 1970
6. const ticksBetweenYear1And1970 = 62135596800000;
7. now += ticksBetweenYear1And1970
8. // Convert the value in now from ms to 100 ns
9. now \*= 10000;
10. // Return the result
11. return now;

}

1. Add a closing brace, **}** to the end of the **checkStockAvailable** function, after the **getCurrentTimeInNetTicks** function.

**Note**: The complete code for the stored procedure is available in the **CheckStockAvailable.txt** file, in the **E:\Labfiles\Lab03** folder.

1. Click **Save** to save the stored procedure.

**Task 2: Test the CheckStockAvailable Stored Procedure**

1. In the Azure portal, select the **Data** collection, and then click **New SQL Query**.
2. Run the following query. This query should return the details of product 788. This product is a mountain bike. Make a note of the value of the **quantityinstock** property:

SELECT \* FROM c WHERE c.id = "788"

1. Return to the **CheckStockAvailable** stored procedure, and then click **Execute**.
2. On the **Input parameters** blade, set **Partition key value** to **Mountain Bikes** (this is a subcategory name, which is used to partition products).
   * Provide **788** as the product id parameter. This is the product ID of one type of mountain bike.
   * Provide **Mountain 300, Black, 48** as the product name parameter. This is the product name of the mountain bike.
   * Provide **3** as the order quantity parameter. This is the number of bikes required.
   * Provide **99** as the customer id parameter. This is the customer ID.
3. Run the stored procedure. You should see a result message similar to the following. This is the JSON data for the response body generated by the stored procedure:

"{message: \"items in stock\", allocated: 3, latestprice:1079.99}"

1. Re-run to the SQL Query you entered earlier. Verify that the value of the **quantityinstock** property has decreased by 3.
2. Run the stored procedure again, but this time change the value of the order quantity parameter from **3** to **300**.. This time, the stored procedure has attempted to reserve more mountain bikes than are in stock and should have created a backorder. You should see a response message similar to this (the number allocated may vary, depending on how many times you have performed the previous steps):

"{message: \"backorder created\", backorderid: \"638c2d72-3118-f848-4b83-05bc3138e7b2\", allocated: 122, latestprice:1079.99}"

1. Re-run to the SQL Query you entered earlier. Verify that the value of the **quantityinstock** property has dropped to 0.
2. Change the query as shown below and then click **Execute Query** again:

SELECT \* FROM c WHERE c.doctype = "ProductBackorder"

1. Verify that the query fetches the **ProductBackorder** document created by the stored procedure. The document **id**should match that shown in the response message of the stored procedure:
2. [
3. {
4. "doctype": "ProductBackorder",
5. "partitionkey": "Mountain Bikes",
6. "customerid": "99",
7. "productid": "788",
8. "productname": "Mountain-300 Black, 48",
9. "productnumber": "BK-M47B-48",
10. "numberonbackorder": 178,
11. "backorderstatus": "In progress",
12. "backorderdate": 636668966127900000,
13. "id": "638c2d72-3118-f848-4b83-05bc3138e7b2",
14. "\_rid": "CE1MAMJEDABRAwAAAAAAAA==",
15. "\_self": "dbs/CE1MAA==/colls/CE1MAMJEDAA=/docs/CE1MAMJEDABRAwAAAAAAAA==/",
16. "\_etag": "\"0000f705-0000-0000-0000-5b45c7e40000\"",
17. "\_attachments": "attachments/",
18. "\_ts": 1531299812
19. }

]

1. Run the stored procedure again, but this time change the value of the product ID parameter from **788** to **7888** (a non-existent product). The stored procedure should throw an error similar to the error shown below, with a message specifying that the product is not available:

{"code":400,"body":"{\"code\":\"BadRequest\",\"message\":\"Message: {\\\"Errors\\\":[\\\"Encountered exception while executing function. Exception = Error: Product Mountain 300, Black, 48 is not available\\\\r\\\\nStack trace: Error: Product Mountain 300, Black, 48 is not available\\\\n at Anonymous function (Test.js:15:13)\\\\n at Anonymous function (Test.js:608:29)\\\"]}\\r\\nActivityId: 0ca5911e-f5e1-4391-8e1a-5758ac824e22, Request URI: /apps/3666e0f0-ed7d-4e2e-8558-f69907b4ff5d/services/2ddcc7a4-a41f-456d-b304-cff7a21c84e0/partitions/a71482ca-4b9a-4c7b-b0cd-e5474963bc97/replicas/131751096055871999p, RequestStats: , SDK: Microsoft.Azure.Documents.Common/2.0.0.0\"}","activityId":"0ca5911e-f5e1-4391-8e1a-5758ac824e22"}

**Task 3: Add Stored Procedure Functionality to the Repository**

1. Using Visual Studio open the the **Adventure-Works.sln** solution in **E:\Labfiles\Lab03\Starter\Exercise03\Adventure-Works**. This solution contains an extended version of the app from Exercise 2, with the addition of a controller and views for managing orders and backorders.
2. Edit the **App.config** file. Set the **EndpointUrl** and **PrimaryKey** settings to the values you recorded for the Cosmos DB account that you created at the beginning of the lab. Confirm that the values of the **Database** and **Collection** settings match your Cosmos DB database and collection.
3. Open the **Repository.cs** file, then find the **ExecuteStoredProcAsync** method at the end of the **Repository<T>** class. The purpose of this method is to run a stored procedure, The name of the stored procedure is passed in as the **storedProc** parameter, and the partition key that defines the scope of the stored procedure is specified in the **partitionkey** parameter. Any arguments to be passed to the stored procedure are held in the **paramList** array.
4. After the comment **TODO: Get the URI of the stored procedure**, add the following statement:

Uri storedProcUri = UriFactory.CreateStoredProcedureUri(database, collection, storedProc);

1. After the comment **TODO: Specify the partition identified by the partitionKey parameter**, add the following code. You specify the partition using the **PartionKey** option in the request options header sent to Cosmos DB when it runs the stored procedure:
2. var options = partitionKey == null ? null : new RequestOptions
3. {
4. PartitionKey = new PartitionKey(partitionKey)

};

1. After the comment **TODO: Run the stored procedure**, add the following statement. This statement invokes the stored procedure, passing in any parameters expected by the stored procedure:

var storedProcResults = await client.ExecuteStoredProcedureAsync<dynamic>(storedProcUri, options, paramList);

1. Change the statement after the comment **TODO: Pass back the results of the stored procedure** as follows. This statement passes the body of the response message back to the caller. This message will contain a JSON object indicating whether a backorder was created, the number of items allocated, and the latest price of the product, as shown in the previous exercise:

return storedProcResults.Response;

**Task 4: Call the CheckStockAvailable Stored Procedure From the App Using the Repository**

1. In the **Controllers** folder, open the **OrdersController.cs** file. The code in this file implements a controller that enables a customer to view and manage orders.
2. Expand the **helper methods** region at the start of the class and examine the **GetOrdersForCustomerAsync** method. This method uses the **ShoppingCartOrder** repository to retrieve the orders for a specified customer and cache them in the session:
3. private async Task<List<ShoppingCartOrder>> GetOrdersForCustomerAsync(Customer customer)
4. {
5. // Find the orders for the customer
6. var orders = await Repository<ShoppingCartOrder>.GetItemsAsync(
7. o => o.DocType == nameof(ShoppingCartOrder) && o.IsShoppingCartOrOrder == Constants.ORDER && o.CustomerID == customer.CustomerID);
8. // Cache the orders in the Session
9. var ordersList = orders.ToList();
10. Session["orders"] = ordersList;
11. // Return the list of orders
12. return ordersList;

}

**Note**: There is a similar method named **GetBackordersForCustomerAsync** which retrieves and caches any backorders that the customer might have.

1. Find the **ViewOrdersAsync** method in the **OrdersController** class. This method is similar to that used to display shopping cart data, expect that it displays orders (and fetches any backorders).
2. // Method that fetches and displays the orders for the customer
3. [ActionName("ViewOrders")]
4. [HttpGet]
5. public async Task<ActionResult> ViewOrdersAsync()
6. {
7. // Check whether the user has logged in
8. if (Session["customer"] == null)
9. {
10. // If not, then get them to log in
11. Session["ReturnData"] = new { ReturnURL = "ViewOrders", Controller = "Orders" };
12. return View("Login");
13. }
14. // Retrieve the orders for the customer
15. var orders = await GetOrdersForCustomerAsync(Session["customer"] as Customer);
16. // Fetch and cache the backorders as well
17. await GetBackordersForCustomerAsync(Session["customer"] as Customer);
18. // Display the orders
19. return View(orders);

}

1. Open the **ViewOrders** view in the **Views/Orders** folder. This is the view generated by the **ViewOrdersAsync** method. It iterates through the list of orders in the input model and displays the order ID, date placed, cost, and status of the order. The view also provides two buttons; **Cancel** which the user can click to cancel an order that hasn't yet been shipped (you will implement this functionality later in the lab), and **Details** which will display the line items in the order by calling the **OrderDetails** action of the **Orders** controller (you can examine the code for this action and the corresponding **OrderDetails** view in your own time).
2. Open the **Global.asax.cs** file. After the comment **TODO: Add ProductBackorder repository**, add the following statement:

Repository<ProductBackorder>.Initialize(ConfigurationManager.AppSettings["Collection"]);

**Note**: Orders are stored in the **ShoppingCartOrder** repository that you created earlier.

1. Open the **ShoppingCartController.cs** file in the **Controllers** folder. This is the controller that you created in exercise 2. At the end of the controller, find the **PlaceOrderAsync** method. This is the method that runs when the user clicks **Place Order** when viewing their shopping cart. Most of the code in this method is currently missing.
2. After the comment **TODO: Create a dictionary for tracking whether backorders are created (if insufficient stock is currently available for any order item)**, add the following statement. The dictionary will be used to hold the product name and backorder ID for any backorders created. This information will be used to display a message to the customer when the order has been placed:

var backorderDetails = new Dictionary<string, string>();

1. After the two-line comment starting **TODO: Use a stored procedure to check that sufficient stock is available for each item, allocate the stock for this order, or create a backorder if necessary**, add the following code. The first statement uses the **Product** repository to run the **CheckStockAvailable** stored procedure. The product subcategory is the partion key. The remaining parameters are passed to the stored procedure by the repository. The second statement parses and deserializes the response from the stored procedure; remember from the earlier tasks thet the response body is a JSON object containing fields named **message**, **backorderid**, **allocated**, and **latestprice**:
2. var response = await Repository<Product>.ExecuteStoredProcAsync("CheckStockAvailable", item.Subcategory, item.ProductID, item.ProductName, item.NumberInCartOrOrdered, (Session["customer"] as Customer).CustomerID);

var responseData = JsonConvert.DeserializeAnonymousType(response, new { message = "", backorderid = "", allocated = 0, latestprice = 0.0M });

1. Add the following block of code after the comment **TODO: Check for backordered items**. If a backorder was created, the backorder ID and product name are added to the **backorderDetails** dictionary. The backorder ID is also recorded in the order line item, and the number on backorder is calculated and also added to the order line item:
2. if ((responseData as dynamic).backorderid != null)
3. {
4. backorderDetails.Add(item.ProductName, (responseData as dynamic).backorderid);
5. item.NumberOnBackorder = item.NumberInCartOrOrdered - (responseData as dynamic).allocated;
6. item.BackorderReference = (responseData as dynamic).backorderid;

}

1. After the comment **TODO: Update the order item with the most up to date cost of the item, and the number allocated**, add the following statements:
2. item.NumberInCartOrOrdered = (responseData as dynamic).allocated;
3. item.UnitCost = (responseData as dynamic).latestprice;

item.LineItemTotalCost = (responseData as dynamic).allocated \* (responseData as dynamic).latestprice;

1. After the comment **TODO: Mark the order as a confirmed order**, add the following line of code. This statement changes the type of the document from **ShoppingCart** to **Order**:

shoppingCart.IsShoppingCartOrOrder = Constants.ORDER;

1. After the comment **TODO: Save the order to the database**, add the following code. This code calls the **UpsertItemAsync** method of the **ShoppingCartOrder** repository to save the order (including the order line items) to the database. If the dictionary indicates that any backorders were created, the code generates a message listing the products and backorder IDs which will be displayed to the customer. The shopping cart is then removed from the cache.
2. if (await Repository<ShoppingCartOrder>.UpsertItemAsync(shoppingCart))
3. {
4. StringBuilder message = new StringBuilder();
5. message.Append("Order placed.");
6. foreach (var backorder in backorderDetails)
7. {
8. message.Append($"\\nBackorder created for {backorder.Key}: Reference {backorder.Value}");
9. }
10. TempData["itemMessage"] = message.ToString();
12. // Remove the shopping cart from the Session cache as it is no longer required
13. Session.Remove("shoppingCart");
14. }
15. else
16. {
17. TempData["itemMessage"] = "Order not placed";

}

1. In the **catch** block that handles the **DocumentClientException** exception, after the comment **TODO: In the event of an exception, return to the ViewCart display and show the error message**, add the code shown below. Remember that if the stored procedure fails it throws an error. This code catches the error as a **DocumentClientException** exception. The error message from the stored procedure contains a lot of information about the location of the error in the stored procedure code (see the earlier task in this exercise, when testing the stored procedure with a non-existent product). While useful for debugging, this information is irrelevant to the customer. The most important part of the message is the contents of the Error object thrown by the stored procedure; this contains a message such as **Product mountain bike not found**, or **Unable to create backorder. Please retry**. The code below uses a regular expression to isolate this content and extract it from the exception, and then display it to the customer. Remember that if an exception occurs while running a stored procedure, any changes made by the stored procedure are discarded.
2. Regex errMsgPattern = new Regex(@".\*(?<error>Error:.\*)\\r\\nStack.\*");
3. Match errMsgMatch = errMsgPattern.Match(dce.Message);
4. if (errMsgMatch.Success)
5. {
6. TempData["itemMessage"] = errMsgMatch.Groups["error"].Value;
7. }
8. else
9. {
10. TempData["itemMessage"] = dce.Message;
11. }

return RedirectToAction("ViewCart", "ShoppingCart");

1. Build and run the web application. Browse the catalog and add items to the shopping cart. On the **Shopping Cart**page, click **Place Order**. Verify that the order is created and is displayed on the **Orders** page. Observe that some details, such as the date placed, cost, and status of the order have not been set yet; you will do this using a trigger that fires when the shopping cart is converted into an order in the database, in the next exercise.
2. Place another order that generates a backorder (attempt to buy 5000 items of a specific product). When you save the order, the **Order placed** message should include a backorder reference. When you view the details of the order, you should see that a quantity (5000 minus the number that was in stock) is on backorder, together with a link for the backorder reference. Click the link and examine the details of the backorder.
3. Close the application and return to Data Explorer in the Azure Portal. Run the following SQL query. Replace **~n~** with the customer ID that you used to place the orders. Verify that your orders appear:

SELECT \* FROM c WHERE c.doctype = "ShoppingCartOrder" AND c.partitionkey = "~n~"

1. Run the following query to find any backorders that have been created:

SELECT \* FROM c WHERE c.doctype = "ProductBackorder"

**Result**: At the end of this exercise, you should have implemented a stored procedure to manage stock levels and create backorders when a customer places an order.

**Exercise 4: Maintaining Data and Auditing Orders and Backorders**

**Exercise 4 Scenario**

When an order is created from items in the shopping cart, you need to calculate the total cost of the order and apply any discounts to which the customer might be entitled. You will create a pre-trigger for orders that performs this task, and also fills in the remaining details of the order that could not be completed earlier.

The app enables a customer to cancel orders and backorders that have not been shipped. For accounting and audit purposes, these changes need to be recorded in an audit trail. You will create a post-trigger that creates audit documents when an order or backorder is cancelled.

**Note**: In this exercise, you will extend the functionality of the Adventure-Works web application. You can either continue using the solution from Exercise 3, or open the **Adventure-Works** solution in the **E:\Labs\Lab03\Starter\Exercise04\Adventure-Works folder**. These instructions assume that you use the solution in the **E:\Labs\Lab03\Starter\Exercise04\Adventure-Works** folder.

The main tasks for this exercise are as follows:

1. Create a pre-trigger to complete details of an order.
2. Create a post-trigger to maintain an audit trail.
3. Extend the application to work with triggers.
4. Test the triggers.
5. Close down the lab.

**Task 1: Create a Pre-Trigger that Completes the Details of an Order**

1. Using the Azure Portal, create a new trigger in the **Data** collection.
   * Set the Id of the new trigger to **CompleteOrderDetails**.
   * Set the **Trigger Type** to **Pre**.
   * Set the **Trigger Operation** to **All**.
2. Add the following JavaScript statements to the body of the trigger function. In a pre-trigger, the document being inserted, updated, or deleted is available in the body of the HTTP request which is part of the context for the JavaScript function:
3. var context = getContext();
4. var request = context.getRequest();
5. var collection = context.getCollection();

var documentBeingUpdated = request.getBody();

1. Add the following code to the function. You only want this trigger to fire when an shopping cart is being converted to an order (an upsert), rather than when the item is being created or deleted. Note that **upsert** is not one of the options available in the **Trigger Operation** list, which is why the check is being performed by using code:
2. // If the operation being performed is not an upsert, then ignore this trigger
3. if (request.getOperationType() != "Upsert") {
4. return;

}

1. Add the following JavaScript block that verifies that the trigger is being called against a **ShoppingCartOrder**document, and not some other type of data:
2. // Verify the type of the document
3. if (!("doctype" in documentBeingUpdated) && documentBeingUpdated.doctype != "ShoppingCartOrder") {
4. throw new Error("Wrong type of document");

}

1. Add the code shown below to the function. This block determines how many items are in the order, and the total cost before applying any discounts:
2. // Accumulator for counting the number of individual items in the order
3. var numberOfItemsInOrder = 0;
4. // Accumulator for counting the total cost of the order (before any customer discount)
5. var costOfItemsInOrder = 0;
6. // Iterate through the order items in the shopping cart
7. documentBeingUpdated.orderitems.forEach(function (orderitem) {
8. // Calculate the total number of items in the order, and the cost of these items
9. numberOfItemsInOrder += orderitem.numberincartorordered;
10. costOfItemsInOrder += orderitem.lineitemtotalcost;

});

1. Add the following code to the function. This code retrieves the discount rate for the customer. It aborts the trigger (and rolls back any changes made) if an error occurs or the customer doesn't exist:
2. // Find the discountrate for the customer
3. var isAccepted = collection.queryDocuments(collection.getSelfLink(), "SELECT \* FROM c WHERE c.doctype = 'Customer' AND c.partitionkey = '" + documentBeingUpdated.partitionkey + "'", {}, function(err, customerDocs, options){
4. // Report any errors
5. if (err) {
6. throw new Error(err);
7. }
8. // If the specified customer was not found in the collection throw an error with a suitable message
9. if (customerDocs.length < 1) {
10. throw new Error("Customer " + documentBeingUpdated.partitionkey + " not found");

}

1. Append the code shown below to the function. This code sets the fields in the order and calculates the cost using the customer's discount rate. This rate is specified as a percentage in between 0 and 100 in the customer document:
2. // Extract the discount rate from the customer document
3. var discountrate = customerDocs[0].discountrate;
4. // Apply the discount to the order
5. documentBeingUpdated.numberofitems = numberOfItemsInOrder;
6. documentBeingUpdated.itemscost = costOfItemsInOrder;
7. documentBeingUpdated.customerdiscountrate = discountrate;
8. documentBeingUpdated.totalcost = documentBeingUpdated.itemscost \* (100 - discountrate) / 100;
9. // Set the order status, date placed, and date last updated
10. documentBeingUpdated.orderstatus = "In progress";
11. documentBeingUpdated.dateplaced = getCurrentTimeInNetTicks();
12. documentBeingUpdated.lastupdated = getCurrentTimeInNetTicks();

});

1. Add the function **getCurrentTimeInNetTicks** to the trigger:
2. function getCurrentTimeInNetTicks() {
3. // Get the current time
4. var now = new Date().getTime();
5. // Adjust for 1st Jan 1970
6. const ticksBetweenYear1And1970 = 62135596800000;
7. now += ticksBetweenYear1And1970
8. // Convert the value in now from ms to 100 ns
9. now \*= 10000;
10. // Return the result
11. return now;

}

1. Add the following code to complete the trigger. Note that the final statement changes the body of the HTTP request that invoked the trigger to reflect the changes made to the document. If you don't do this, the original document will be used, and you will lose the changes made by the trigger.
2. // If the trigger is out of runtime, throw an error
3. if (!isAccepted) {
4. throw new Error('Unable to get customer details');
5. }
6. // Save the changes made to the order

request.setBody(documentBeingUpdated);

**Note**: The complete code for the trigger is available in the file **CompleteOrderDetails.txt** in the **E:\Labfiles\Lab03**folder.

1. Save the trigger when you have finished editing.

**Task 2: Create a Post-trigger that Audits Changes Made to Orders and Backorders**

1. Add a new trigger to the **Data** collection
   * Set the trigger Id to **AuditOrder**.
   * Set the **Trigger Type** to **Post**.
   * Set the **Trigger Operation** to **All**.
2. Add the following JavaScript statements to the body of the trigger function. In a post-trigger, the document has already been written to the database, but its contents are available in the HTTP response message which you can access using the **getResponse** API function:
3. var context = getContext();
4. var response = context.getResponse();
5. var collection = context.getCollection();

var documentBeingUpdated = response.getBody();

1. Add the following code to the function to verify that the document being processed is either an order or a backorder.
2. // Verify the type of the document
3. if (!("doctype" in documentBeingUpdated) && documentBeingUpdated.doctype != "ShoppingCartOrder" && documentBeingUpdated.docType != "ProductBackorder") {
4. throw new Error("Wrong type of document");

}

1. Add the following code that creates a new audit document using the information in the order or backorder. Notice that the **partitionkey** field is set to the same value as that of the document being updated. This will cause the document to be stored in the same partition; you cannot add documents to other partitions in a trigger:
2. // Create a document with the audit details
3. var auditDocument = {
4. partitionkey: documentBeingUpdated.partitionkey,
5. doctype: "AuditDocument",
6. doctypebeingaudited: documentBeingUpdated.doctype,
7. documentbeingaudited: documentBeingUpdated.id,
8. datechanged: getCurrentTimeInNetTicks(),
9. statuschangedto: documentBeingUpdated.doctype == "ShoppingCartOrder" ? documentBeingUpdated.orderstatus : documentBeingUpdated.backorderstatus,
10. customerid: documentBeingUpdated.doctype == "ShoppingCartOrder" ? documentBeingUpdated.partitionkey : documentBeingUpdated.customerid

};

1. Add the following code that saves the audit document to the database:
2. // Add the audit document to the database
3. var isAccepted = collection.createDocument(collection.getSelfLink(), auditDocument);
4. // If the trigger is out of runtime, throw an error
5. if (!isAccepted) {
6. throw new Error('Unable to create audit record');

}

1. Add the **getCurrentTimeInNetTicks** function shown below to the trigger. This is the same function that was used by the stored procedure. Cosmos DB does not currently permit triggers and stored procedures to share code (you cannot call a UDF from a stored procedure or trigger either):
2. function getCurrentTimeInNetTicks() {
3. // Get the current time
4. var now = new Date().getTime();
5. // Adjust for 1st Jan 1970
6. const ticksBetweenYear1And1970 = 62135596800000;
7. now += ticksBetweenYear1And1970
8. // Convert the value in now from ms to 100 ns
9. now \*= 10000;
10. // Return the result
11. return now;

}

**Note**: The complete code for the trigger is available in the file **AuditOrder.txt** in the **E:\Labfiles\Lab03** folder.

1. Save the trigger when you have finished editing.

**Task 3: Add Trigger Functionality to the Repository**

1. Using Visual Studio open the the **Adventure-Works.sln** solution in **E:\Labfiles\Lab03\Starter\Exercise04\Adventure-Works**. This solution contains a finished version of the app from Exercise 3.
2. Edit the **App.config** file. Set the **EndpointUrl** and **PrimaryKey** settings to the values you recorded for the Cosmos DB account that you created at the beginning of the lab. Confirm that the values of the **Database** and **Collection** settings match your Cosmos DB database and collection.
3. Open the **Repository.cs** file and locate the **UpsertItemAsync** method. This method runs when the app attempts to perform an upsert operation on a document.
4. Examine the parameters for this method; as well as the document being **upserted**, the method takes two optional lists containing the names of pre and post triggers, and a **ttl** value:
5. // Insert/update an item in the collection and optionally provide lists of pre- and post-triggers to run
6. public static async Task<bool> UpsertItemAsync(T item, List<string> preTriggers = null, List<string> postTriggers = null, int ttl = -1)
7. {
8. ...

}

1. After the comment **TODO: Set the TTL of the document**, add the following statement:

item.TimeToLive = ttl;

1. After the comment **TODO: Specify the triggers to be invoked (if any)**, add the following code. This code creates a **RequestOptions** object that specifies the names of the triggers passed in to the method:
2. var options = new RequestOptions
3. {
4. PreTriggerInclude = preTriggers,
5. PostTriggerInclude = postTriggers

};

1. Replace the statement after the comment **TODO: Add/replace the document in the collection and invoke the triggers** with the following code. This code calls the UpsertDocumentAsync method but now includes the **RequestOptions** object that specifies the triggers to be run as part of the request:
2. var response = await client.UpsertDocumentAsync(collectionUri, item, options);

**Task 4: Invoke the Triggers**

1. Open the **ShoppingCartController.cs** file in the **Controllers** folder and find the **PlaceOrderAsync** method at the end of the class.
2. In this method, find the following statement:
3. // Save the order to the database

if (await Repository<ShoppingCartOrder>.UpsertItemAsync(shoppingCart))

1. Update the call to the **UpsertItemAsync** method of the **ShoppingCartOrder** repository to specify that the **CompleteOrderDetails** pre-trigger should be run as part of the operation:

if (await Repository<ShoppingCartOrder>.UpsertItemAsync(shoppingCart, preTriggers: new List<string> { "CompleteOrderDetails" }))

1. Open the **OrdersController.cs** file in the **Controllers** folder and find the **CancelOrderAsync** method. As its name suggests, this method runs when the customer attempts to cancel an order.
2. In this method, find the following comment and line of code:
3. // TODO: Save the order, and fire AuditOrder post-trigger

if (1==0)

1. Replace the **if** statement with the following code. This statement calls the **UpsertItemAsync** method and specifies that the **AuditOrder** post-trigger should run as part of the operation:

if (await (Repository<ShoppingCartOrder>.UpsertItemAsync(orderToCancel, postTriggers: new List<string> { "AuditOrder" })))

1. Still in the **CancelOrderAsync** method, find the comment **TODO: Save the backorder, and fire the AuditOrder post-trigger**. After this comment, add the following code:
2. if (await (Repository<ProductBackorder>.UpsertItemAsync(backorderToCancel, postTriggers: new List<string> { "AuditOrder" })))
3. {
4. message.Append($"\nBackorder {orderItem.BackorderReference} cancelled");

}

1. Find the **CancelBackorderAsync** method at the end of the class. In this method, after the comment **TODO: Save the backorder, and fire the AuditOrder post-trigger**, add the following code:
2. if (await (Repository<ProductBackorder>.UpsertItemAsync(backorderToCancel, postTriggers: new List<string> { "AuditOrder" })))
3. {
4. TempData["itemMessage"] = $"Backorder {backorderID} cancelled";

}

1. Build and run the web application. Browse the catalog and add items to the shopping cart, and create a new order. When the **Orders** page appears, verify that the date placed, cost, and order status are now filled in.
2. Place another order that generates a backorder (attempt to buy 5000 items of a specific product). When you save the order, the **Order placed** message should include a backorder reference. When you view the details of the order, you should see that a quantity (5000 minus the number that was in stock) is on backorder, together with a link for the backorder reference. Click the link and examine the details of the backorder.
3. Close the application and return to Data Explorer in the Azure Portal. Run the following SQL query. Replace **~n~** with the customer ID that you used to place the orders. Verify that your orders appear:

SELECT \* FROM c WHERE c.doctype = "ShoppingCartOrder" AND c.partitionkey = "~n~"

1. Run the following query to find any backorders that have been created:

SELECT \* FROM c WHERE c.doctype = "ProductBackorder"

**Task 5: Lab close down**

1. Close any open instances of Visual Studio and File Explorer.
2. To reduce your costs delete the **20777\_Mod03** resource group containing your Cosmos DB database and storage account.

**Result**: In this lab, you have used server-side programming to add functionality to the online sales applications.

**Lab: Writing User-Defined Functions, Stored Procedures and Triggers**

**Exercise 1: Redesigning the Document Structure and Partitioning Strategy**

**Task 1: Prepare the Environment**

1. Ensure that the **MT17B-WS2016-NAT** and **20777A-LON-DEV** virtual machines are running, and then log on to **20777A-LON-DEV** as **Administrator** with the password **Pa55w.rd**.
2. On the toolbar, click **File Explorer**.
3. In File Explorer, navigate to **E:\Labfiles\Lab03\Starter**, right-click **Setup.cmd**, and then click **Run as administrator**.
4. Wait for the script to finish.
5. On the toolbar, click **Internet Explorer**.
6. In Internet Explorer, go to [**http://portal.azure.com**](http://portal.azure.com/), and sign in using the Microsoft account that is associated with your Azure Learning Pass subscription.
7. In the Azure Portal, click **+ Create a resource**.
8. On the **New** blade, in the search box, type **storage account**, and then press Enter.
9. On the **Everything** blade, click **Storage account - blob, file, table, queue**.
10. On the **Storage account - blob, file, table, queue** blade, click **Create**.
11. On the **Create storage account** blade, under the **Resource group** box, click **Create new**, type **20777\_Mod03**, and then click **OK**.
12. In the **Storage account name** box, type **20777blob<*your name><the day*>**, for example, **20777blobjohn31**.
13. In the **Location** drop-down list, click the region closest to your current location.
14. In the **Account kind** list, click **BlobStorage**.
15. In the **Replication** list, click **Locally-redundant storage (LRS)**.
16. Click **Review + create**, and then click **Create**.
17. In the Azure Portal, click **All resources**, and then click **20777blob<*your name><the day*>**.
18. On the **20777blob<*your name><the day*>** blade, under **Blob Service**, click **Blobs**.
19. On the **Blobs** blade, click **+ Container**.
20. In the **New container** dialog box, in the **Name** box, type **20777-mod3-blobs**.
21. In the **Public access level** drop-down list, click **Blob (anonymous read access for blobs only)**, and then click **OK**.
22. On the **20777blob<*your name><the day*>** blade, under **Settings**, click **Access keys**.
23. Under **key1**, make a note of the **Connection string** value.
24. In the left pane of the Azure portal, click **+ Create a resource**.
25. On the **New** blade, in the search box, type **Cosmos DB**, and then press Enter.
26. On the **Everything** blade, click **Azure Cosmos DB**, and then click **Create**.
27. On the **Azure Cosmos DB** blade, in the **Resource Group**, drop-down list, click **20777\_Mod03**.
28. In the **Account name** box, type **20777-sql-<*your name>-<the day*>**, for example, **20777-sql-john-31**.
29. In the **API** drop-down list, click **Core (SQL)**.
30. In the **Location** drop-down list, click the region closest to your current location.
31. Click **Review + create**, and then click **Create**.
32. Wait for the Azure Cosmos DB to be created—this could take a few minutes.
33. In the Azure Portal, click **All resources**, and then click **20777-sql-<*your name>-<the day*>**.
34. On the **20777-sql-<*your name>-<the day*>** blade, click **Data Explorer**.
35. In the **SQL API** pane, click **New Database**.
36. On the **New Database** blade, in the **Database id** box, type **Adventure-Works**, and then click **OK**.
37. On the **20777a-sql-<*your name>-<the day*>** blade, under **Settings**, click **Keys**.
38. Make a note of the **URI**, and **PRIMARY KEY** values.

**Task 2: Analyze the Technical Requirements for Document Partitioning**

No detailed steps for this task. Read the description for the task in **20777A\_LAB\_03.md**.

**Task 3: Implement the Different Types of Documents**

1. On the Start menu, click **Visual Studio 2017**.
2. On the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, in the **File name** box, type **E:\Labfiles\Lab03\Starter\Exercise01\Adventure-Works\Adventure-Works.sln**, and then click **Open**.
4. In the **Security Warning for Adventure-Works** dialog box, clear the **Ask me for every project in this solution** check box, click **OK**.
5. In Solution Explorer, double-click **Web.config**.
6. In App.config, in the **Value** attribute of the **EndpointUrl** key, paste the **URI** value you noted earlier, replacing the text **<URL>**.
7. In the **Value** attribute of the **PrimaryKey** key, paste the **PRIMARY KEY** value you noted earlier, replacing the text **<PRIMARY KEY>**.
8. In Solution Explorer, expand **Models**, and then double-click **DocumentType.cs**.
9. In DocumentType.cs, after the comment **// TODO: doctype distinguishes the different types of documents in the same collection**, type the following code:
10. [JsonProperty("doctype")]

public string DocType { get; set; }

1. After the comment **// TODO: Initialize the doctype property**, type the following code:

this.DocType = this.GetType().Name;

1. After the comment **// TODO: ttl of document. By default, ttl is disabled, but can be changed by setting it to a positive integer value (number of seconds)**, type the following code:
2. [JsonProperty("ttl")]

public int TimeToLive { get; set; } = -1;

1. In Solution Explorer, under **Models**, double-click **Customer.cs**.
2. In Customer.cs, replace:
3. public class CustomerName
4. {
5. // TODO: Define properties

}

with:

public class CustomerName

{

[JsonProperty("title")]

public string Title { get; set; }

[JsonProperty("firstname")]

public string FirstName { get; set; }

[JsonProperty("middlename")]

public string MiddleName { get; set; }

[JsonProperty("lastname")]

public string LastName { get; set; }

}

1. In Customer.cs, replace:
2. public class Customer : DocumentType
3. {
4. // TODO: Define properties

}

with:

public class Customer : DocumentType

{

[JsonProperty("partitionkey")]

public string CustomerID { get; set; }

[JsonProperty("name")]

public CustomerName CustomerName { get; set; }

[JsonProperty("discountrate")]

public int DiscountRate { get; set; }

}

1. In Solution Explorer, under **Models**, double-click **ShoppingCartOrder.cs**.
2. In ShoppingCartOrder.cs, replace:
3. public class OrderItem
4. {
5. // TODO: Define properties

}

with:

public class OrderItem

{

[JsonProperty("productnumber")]

public string ProductNumber { get; set; }

[JsonProperty("productname")]

public string ProductName { get; set; }

[JsonProperty("productid")]

public string ProductID { get; set; }

[JsonProperty("subcategory")]

public string Subcategory { get; set; }

[JsonProperty("numberincartorordered")]

public int NumberInCartOrOrdered { get; set; }

[JsonProperty("numberonbackorder")]

public int NumberOnBackorder { get; set; }

[JsonProperty("backorderreference")]

public string BackorderReference { get; set; }

[JsonProperty("unitcost")]

public decimal UnitCost { get; set; }

[JsonProperty("lineitemtotalcost")]

public decimal LineItemTotalCost { get; set; }

}

1. In ShoppingCartOrder.cs, replace:
2. public class ShoppingCartOrder : DocumentType
3. {
4. // TODO: Define properties

}

with:

public class ShoppingCartOrder : DocumentType

{

[JsonProperty("id")]

public string ShoppingCartOrderID { get; set; }

[JsonProperty("partitionkey")]

public string CustomerID { get; set; }

[JsonProperty("isshoppingcartororder")]

public string IsShoppingCartOrOrder { get; set; }

[JsonProperty("orderitems")]

public List<OrderItem> OrderItems { get; set; }

[JsonProperty("numberofitems")]

public int NumberOfItems { get; set; }

[JsonProperty("itemscost")]

public decimal ItemsCost { get; set; }

[JsonProperty("customerdiscountrate")]

public int CustomerDiscountRate { get; set; } // Indicates % discount

[JsonProperty("totalcost")]

public decimal TotalCost { get; set; }

[JsonProperty("dateplaced")]

public long DatePlaced { get; set; } // Stored as ticks

[JsonProperty("orderstatus")]

public string OrderStatus { get; set; } // "In progress", "Delivered", "Cancelled"

[JsonProperty("lastupdated")]

public long LastUpdated { get; set; } // Also stored as ticks

}

**Task 4: Create a Document Database Using the Cosmos DB SQL API**

1. In Internet Explorer, in the Azure Portal, on the **20777-sql-<*your name>-<the day*>** blade, click **Data Explorer**.
2. In the **SQL API** pane, right-click **Adventure-Works**, and then click **New Collection**.
3. On the **Add Collection** blade, in the **Collection id** box, type **Data**.
4. In the **Partition key** box, type **/partitionkey**.
5. In the **Throughput (400 - 1,000,000 RU/s)** box, type **1000**, and then click **OK**.
6. In Visual Studio, press F5 to run and start the application.
7. In Internet Explorer, verify that the **Adventure-Works Product Catalog** page is displayed.
8. Close the Adventure-Works Product Catalog page to stop the application.
9. Close Visual Studio 2015.

**Task 5: Import the Product and Customer Data**

1. On the Start menu, click **Visual Studio 2017**.
2. On the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, in the **File name** box, type **E:\Labfiles\Lab03\Starter\Exercise01\MigrateProductData\MigrateProductData.sln**, and then click **Open**.
4. In the **Security Warning for Adventure-Works** dialog box, clear the **Ask me for every project in this solution** check box, click **OK**.
5. In Solution Explorer, double-click **App.config**.
6. In App.config, in the **Value** attribute of the **EndpointUrl** key, paste the **URI** value you noted earlier, replacing the text <**URL**>.
7. In the **Value** attribute of the **PrimaryKey** key, paste the **PRIMARY KEY** value you noted earlier, replacing the text **<PRIMARY KEY>**.
8. In the **Value** attribute of the **BlobStorageConnectionString** key, paste the **Connection string** value you noted earlier, replacing the text **<CONNECTION STRING>**.
9. In the **Value** attribute of the **BlobContainer** key, delete the text **<BLOB CONTAINER NAME>**, and then type **20777-mod3-blobs**.
10. Press F5 to run the application. Verify that it retrieves the data from SQL Server and uploads the documents to Cosmos DB without reporting any errors.
11. When the application finishes, close Visual Studio.
12. In Internet Explorer, in the **SQL API** pane, expand **Adventure-Works**, expand **Data**, and then click **Documents**.
13. On the **Documents** tab, click on one of the documents in the list to view it.
14. In the **SQL API** pane, click **Upload**.
15. On the **Upload Documents** blade, click the folder icon.
16. In the **Choose File to Upload** dialog box, in the **File name** box, type **E:\Labfiles\Lab03\Data\categories.json**, and then click **Open**.
17. On the **Upload Documents** blade, click **Upload**.
18. Wait for the upload to complete; the result message will report that **38 documents created**.
19. On the **Upload Documents** blade, click the folder icon.
20. In the **Choose File to Upload** dialog box, in the **File name** box, type **E:\Labfiles\Lab03\Data\customers.json**, and then click **Open**.
21. On the **Upload Documents** blade, click **Upload**.
22. Wait for the upload to complete; the result message will report that **100 documents created**.
23. Close the **Upload Documents** blade.
24. Close Visual Studio 2015.

**Exercise 2: Implementing the Shopping Cart Functionality**

**Task 1: Configure the Repository Classes**

1. On the Start menu, click **Visual Studio 2017**.
2. On the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, in the **File name** box, type **E:\Labfiles\Lab03\Starter\Exercise02\Adventure-Works\Adventure-Works.sln**, and then click **Open**.
4. In the **Security Warning for Adventure-Works** dialog box, clear the **Ask me for every project in this solution** check box, click **OK**.
5. In Solution Explorer, double-click **Web.config**.
6. In App.config, in the **Value** attribute of the **EndpointUrl** key, paste the **URI** value you noted earlier, replacing the text **<URL>**.
7. In the **Value** attribute of the **PrimaryKey** key, paste the **PRIMARY KEY** value you noted earlier, replacing the text **<PRIMARY KEY>**.
8. In Solution Explorer, double-click **Global.asax**.
9. In Global.asax.cs, after the comment **// TODO: Add Customer repository**, type the following code:

Repository<Customer>.Initialize(ConfigurationManager.AppSettings["Collection"]);

1. After the comment **// TODO: Add ShoppingCartOrder repository**, type the following code:

Repository<ShoppingCartOrder>.Initialize(ConfigurationManager.AppSettings["Collection"]);

**Task 2: Complete the Customer Controller to Enable Customers to Login and Logout**

1. In Solution Explorer, expand **Controllers**, and then double-click **CustomerController.cs**.
2. In CustomerController.cs, after the comment **// TODO: Use the repository to retrieve the details of the specified customer**, type the following code:

var customer = (await Repository<Customer>.GetItemsAsync(c => c.CustomerID == customerID)).FirstOrDefault();

1. After the comment **// TODO: Validation and authentication using the password etc should go here. Only validation is implemented in this example, the user can enter anything in the password field!**, type the following code:
2. if (customer == null) // No such customer
3. {
4. TempData["invalidUserOrPassword"] = "Invalid username or password";
5. return View();

}

1. After the comment **// TODO: If authentication is successful (it always is in this example), cache the customer details in the session**, type the following code:

Session["customer"] = customer;

1. After the comment **// TODO: Retrieve the current shopping cart for the customer (if there is one)**, type the following code:
2. var shoppingCart = (await Repository<ShoppingCartOrder>.GetItemsAsync(c => c.CustomerID == customerID && c.IsShoppingCartOrOrder == Constants.SHOPPINGCART)).FirstOrDefault();

Session["shoppingCart"] = shoppingCart;

1. Review the **Logout** method at the end of the file.

**Task 3: Implement Functionality to Add Items to the Shopping Cart**

1. In Solution Explorer, under **Controllers**, double-click **ShoppingCartController.cs**.
2. In ShoppingCartController.cs, after the comment **// TODO: Check whether the user has logged in**, type the following code:
3. if (Session["customer"] == null)
4. {
5. // If not, then get them to log in
6. Session["ReturnData"] = new { ReturnURL = "ViewCart", Controller = "ShoppingCart" };
7. return View("Login");

}

1. After the comment **// TODO: Retrieve the shopping cart for the customer**, type the following code:

var shoppingCart = GetShoppingCartForCustomer(Session["customer"] as Customer);

1. After the comment **// TODO: Display the shopping cart**, edit the line:

return View();

so that it reads:

return View(shoppingCart);

1. In ShoppingCartController.cs, after the comment **// TODO: Save the shopping cart using the repository, and then cache it in the session. Set the TTL to 7 days. Note that TTL must be enabled at the collection level for this to work**, type the following code:
2. if (await Repository<ShoppingCartOrder>.UpsertItemAsync(shoppingCart, ttl: 60 \* 60 \* 24 \* 7))
3. {
4. Session["shoppingCart"] = shoppingCart;
5. // Inform the user that the item has been added to the cart
6. TempData["itemMessage"] = "Item added to shopping cart";
7. }
8. else
9. {
10. // If the upsert failed, then inform the user that the item has not been added to the shopping cart
11. TempData["itemMessage"] = "Item not added to shopping cart";

}

1. In Internet Explorer, in the **SQL API** pane, expand **Adventure-Works**, expand **Data**, and then click **Scale & Settings**.
2. On the **Scale & Settings** tab, under **Settings**, under **Time to Live**, click **On (no default)**, and then click **Save**.
3. In Visual Studio, press F5 to run and start the application.
4. On the **Adventure-Works Product Catalog** page, click **Search By Category**.
5. On the first row of search results, click **Details**.
6. On the **Product Details** page, click **Add to Shopping Cart**.
7. On the **Login** page, in the **Customer ID** box, type **10**, and then click **Login**.
8. In the **Message from webpage** dialog box, click **OK**.
9. On the **Shopping Cart** page, click the **Adventure-Works Product Catalog** link.
10. On the **Adventure-Works Product Catalog** page, click **Search By Category**.
11. On the second row of search results, click **Details**.
12. On the **Product Details** page, click **Add to Shopping Cart**.
13. In the **Message from webpage** dialog box, click **OK**.
14. On the **Shopping Cart** page, on the first row, in the **Number Required** box, type **2**, and then click **Update**.
15. In the **Message from webpage** dialog box, click **OK**.
16. On the **Shopping Cart** page, on the second row. click **Delete**.
17. In the **Message from webpage** dialog box, click **OK**.
18. Close the **Shopping Cart** page to stop the application.
19. In Internet Explorer, in the **SQL API** pane, under **Adventure-Works**, right-click **Data**, and then click **New SQL Query**.
20. On the **Query 1** tab, edit the query so that it reads:

SELECT \* FROM c WHERE c.doctype = "ShoppingCartOrder"

1. Click **Execute Query**. One row should be returned, corresponding to the shopping cart that you created.

**Exercise 3: Creating Orders and Backorders**

**Task 1: Create the CheckStockAvailable Stored Procedure**

1. In Internet Explorer, in the **SQL API** pane, under **Adventure-Works**, right-click **Data**, and then click **New Stored Procedure**.
2. On the **New Stored Procedure 1** tab, in the **Stored Procedure Id** box, type **CheckStockAvailable**.
3. In the **Stored Procedure Body** box, delete the sample stored procedure code, and then type the following code:
4. function checkStockAvailable(productID, productname, numRequired, customerID){
5. var collection = getContext().getCollection();
6. var collectionLink = collection.getSelfLink();
7. var response = getContext().getResponse();
8. var isAccepted = collection.queryDocuments(collectionLink, "SELECT \* FROM p WHERE p.doctype = 'Product' AND p.id = '" + productID + "'", {}, function(err, docs, options) {
9. // If an error has occurred, abort the stored procedure
10. if (err) {
11. throw err;
12. }
13. // Check that a doc was found. If not, throw a "not available" error
14. if (!docs || !docs.length) {
15. throw new Error("Product " + productname + " is not available");
16. }
17. else {
18. // If a matching product was found, check for sufficient stock levels
19. var productDoc = docs[0];
20. if (numRequired > productDoc.quantityinstock) {
21. // If there is insufficient stock available, allocate what is there
22. var numOnBackorder = numRequired - productDoc.quantityinstock;
23. updateStockLevel(productDoc, productDoc.quantityinstock);
24. // Create a backorder for the remainder
25. var backorder = {
26. doctype: "ProductBackorder",
27. partitionkey: productDoc.partitionkey,
28. customerid: customerID,
29. productid: productDoc.id,
30. productname: productDoc.productname,
31. productnumber: productDoc.productnumber,
32. numberonbackorder: numOnBackorder,
33. backorderstatus: "In progress",
34. backorderdate: getCurrentTimeInNetTicks()
35. };
36. // Add the backorder to the database.
37. // If an error occurs, the stored procedure will be aborted and changes rolled back
38. isAccepted = collection.createDocument(collectionLink, backorder, function(err, backorderDoc, options){
39. // If an error has occurred, abort the stored procedure
40. if (err) {
41. throw err;
42. }
43. // Set the response body to indicate how many items were allocated, and the details of any backorder
44. response.setBody('{message: "backorder created", backorderid: "' + backorderDoc.id + '", allocated: ' + (numRequired - numOnBackorder) + ', latestprice:' + productDoc.listprice + '}');
45. });
46. if (!isAccepted) {
47. throw new Error('Unable to create backorder. Please retry.');
48. }
49. }
50. else {
51. // If there is sufficient stock available, then allocate it
52. updateStockLevel(productDoc, numRequired);
53. response.setBody('{message: "items in stock", allocated: ' + numRequired + ', latestprice:' + productDoc.listprice + '}');
54. }
55. }
56. });
57. if (!isAccepted) {
58. throw new Error('Unable to check stock level. Please retry.');

}

1. Type the following code, and then click **Save**:
2. function updateStockLevel(productDoc, numRequired) {
3. // Update the stock level
4. productDoc.quantityinstock -= numRequired;
5. // Save the product doc back to the collection, and check for concurrent updates
6. // If an error occurs, the stored procedure will be aborted and changes rolled back
7. isAccepted = collection.replaceDocument(productDoc.\_self, productDoc, {etag: productDoc.Etag});
8. if (!isAccepted) {
9. throw new Error('Unable to update stock level. Please retry.');
10. }
11. }
12. function getCurrentTimeInNetTicks() {
13. // Get the current time
14. var now = new Date().getTime();
15. // Adjust for 1st Jan 1970
16. const ticksBetweenYear1And1970 = 62135596800000;
17. now += ticksBetweenYear1And1970
18. // Convert the value in now from ms to 100 ns
19. now \*= 10000;
20. // Return the result
21. return now;
22. }

}

**Task 2: Test the CheckStockAvailable Stored Procedure**

1. In the Azure portal, in the **SQL API** pane, right-click **Data**, and then click **New SQL Query**.
2. On the **Query 2** tab, edit the query so that it reads the following, and then click **Execute Query**:

SELECT \* FROM c WHERE c.id = "788"

1. In the **Results** pane, note the value of the **quantityinstock** property.
2. On the **CheckStockAvailable** tab, click **Execute**.
3. On the **Input parameters** blade, in the **Partition key value** box, type **Mountain Bikes**.
4. In the **Param** box, type **788**.
5. Click **Add New Param**, in the new parameter box, type **Mountain 300, Black, 48**.
6. Click **Add New Param**, in the new parameter box, type **3**.
7. Click **Add New Param**, in the new parameter box, type **99**, and then click **Execute**.
8. Verify that the stored procedure ran correctly.
9. On the **Query 2** tab, click **Execute Query**. Verify that the value of the **quantityinstock** property has reduced by 3.
10. On the **CheckStockAvailable** tab, click **Execute**.
11. On the **Input parameters** blade, change the value of third parameter from **3** to **300**, and then click **Execute**.
12. Verify that the stored procedure ran correctly.
13. On the **Query 2** tab, click **Execute Query**. Verify that the value of the **quantityinstock** property has reduced to 0.
14. Edit the query so that it reads the following, and then click **Execute Query**:

SELECT \* FROM c WHERE c.doctype = "ProductBackorder"

1. Verify that the **numberonbackorder** property reflects the difference between the **quantityinstock** that you noted earlier and the total number of orders you have placed.
2. On the **CheckStockAvailable** tab, click **Execute**.
3. On the **Input parameters** blade, change the value of first parameter from **788** to **7888**, and then click **Execute**.
4. Verify that an error message is returned by the stored procedure.

**Task 3: Add Stored Procedure Functionality to the Repository**

1. Close any open instances of Visual Studio.
2. On the Start menu, click **Visual Studio 2017**.
3. On the **File** menu, point to **Open**, and then click **Project/Solution**.
4. In the **Open Project** dialog box, in the **File name** box, type **E:\Labfiles\Lab03\Starter\Exercise03\Adventure-Works\Adventure-Works.sln**, and then click **Open**.
5. In the **Security Warning for Adventure-Works** dialog box, clear the **Ask me for every project in this solution** check box, click **OK**.
6. In Solution Explorer, double-click **Web.config**.
7. In App.config, in the **Value** attribute of the **EndpointUrl** key, paste the **URI** value you noted earlier, replacing the text **<URL>**.
8. In the **Value** attribute of the **PrimaryKey** key, paste the **PRIMARY KEY** value you noted earlier, replacing the text **<PRIMARY KEY>**.
9. In Solution Explorer, double-click **Repository.cs**.
10. In Repository.cs, locate the **ExecuteStoredProcAsync** method (line 136).
11. After the comment **// TODO: Get the URI of the stored procedure**, type the following code:

Uri storedProcUri = UriFactory.CreateStoredProcedureUri(database, collection, storedProc);

1. After the comment **// TODO: Specify the partition identified by the partitionKey parameter**, type the following code:
2. var options = partitionKey == null ? null : new RequestOptions
3. {
4. PartitionKey = new PartitionKey(partitionKey)

};

1. After the comment **// TODO: Run the stored procedure**, type the following code:

var storedProcResults = await client.ExecuteStoredProcedureAsync<dynamic>(storedProcUri, options, paramList);

1. After the comment **// TODO: Pass back the results of the stored procedure**, replace:

return new NotImplementedException();

with:

return storedProcResults.Response;

**Task 4: Call the CheckStockAvailable Stored Procedure From the App Using the Repository**

1. In Solution Explorer, double-click **Global.asax**.
2. In Global.asax.cs, after the comment **// TODO: Add ProductBackorder repository**, type the following code:

Repository<ProductBackorder>.Initialize(ConfigurationManager.AppSettings["Collection"]);

1. In Solution Explorer, expand **Controllers**, and then double-click **ShoppingCartController.cs**.
2. In ShoppingCartController.cs, after the comment **// TODO: Create a dictionary for tracking whether backorders are created (if insufficient stock is currently available for any order item)**, type the following code:

var backorderDetails = new Dictionary<string, string>();

1. After the comment **// TODO: Use a stored procedure to check that sufficient stock is available for each item, allocate the stock for this order, or create a backorder if necessary**, type the following code:
2. var response = await Repository<Product>.ExecuteStoredProcAsync("CheckStockAvailable", item.Subcategory, item.ProductID, item.ProductName, item.NumberInCartOrOrdered, (Session["customer"] as Customer).CustomerID);

var responseData = JsonConvert.DeserializeAnonymousType(response, new { message = "", backorderid = "", allocated = 0, latestprice = 0.0M });

1. After the comment **// TODO: Check for backordered items**, type the following code:
2. if ((responseData as dynamic).backorderid != null)
3. {
4. backorderDetails.Add(item.ProductName, (responseData as dynamic).backorderid);
5. item.NumberOnBackorder = item.NumberInCartOrOrdered - (responseData as dynamic).allocated;
6. item.BackorderReference = (responseData as dynamic).backorderid;

}

1. After the comment **// TODO: Update the order item with the most up to date cost of the item, and the number allocated**, type the following code:
2. item.NumberInCartOrOrdered = (responseData as dynamic).allocated;
3. item.UnitCost = (responseData as dynamic).latestprice;

item.LineItemTotalCost = (responseData as dynamic).allocated \* (responseData as dynamic).latestprice;

1. After the comment **// TODO: Mark the order as a confirmed order**, type the following code:

shoppingCart.IsShoppingCartOrOrder = Constants.ORDER;

1. After the comment **// TODO: Save the order to the database**, type the following code:
2. if (await Repository<ShoppingCartOrder>.UpsertItemAsync(shoppingCart))
3. {
4. StringBuilder message = new StringBuilder();
5. message.Append("Order placed.");
6. foreach (var backorder in backorderDetails)
7. {
8. message.Append($"\\nBackorder created for {backorder.Key}: Reference {backorder.Value}");
9. }
10. TempData["itemMessage"] = message.ToString();
11. // Remove the shopping cart from the Session cache as it is no longer required
12. Session.Remove("shoppingCart");
13. }
14. else
15. {
16. TempData["itemMessage"] = "Order not placed";

}

1. After the comment **// TODO: In the event of an exception, return to the ViewCart display and show the error message**, type the following code:
2. Regex errMsgPattern = new Regex(@".\*(?<error>Error:.\*)\\r\\nStack.\*");
3. Match errMsgMatch = errMsgPattern.Match(dce.Message);
4. if (errMsgMatch.Success)
5. {
6. TempData["itemMessage"] = errMsgMatch.Groups["error"].Value;
7. }
8. else
9. {
10. TempData["itemMessage"] = dce.Message;
11. }

return RedirectToAction("ViewCart", "ShoppingCart");

1. Press F5 to run and start the application.
2. On the **Adventure-Works Product Catalog** page, click **Search By Category**.
3. On the first row of search results, click **Details**.
4. On the **Product Details** page, click **Add to Shopping Cart**.
5. On the **Login** page, in the **Customer ID** box, type **20**, and then click **Login**.
6. In the **Message from webpage** dialog box, click **OK**.
7. On the **Shopping Cart** page, click the **Adventure-Works Product Catalog** link.
8. On the **Adventure-Works Product Catalog** page, click **Search By Category**.
9. On the second row of search results, click **Details**.
10. On the **Product Details** page, click **Add to Shopping Cart**.
11. In the **Message from webpage** dialog box, click **OK**.
12. On the **Shopping Cart** page, click **Place Order**.
13. In the **Message from webpage** dialog box, click **OK**.
14. On the **Orders** page, click **Details**, and then click the **Adventure-Works Product Catalog** link.
15. On the **Adventure-Works Product Catalog** page, click **Search By Category**.
16. On the first row of search results, click **Details**.
17. On the **Product Details** page, in the **Number Required** box, type **5000**, and then click **Add to Shopping Cart**.
18. In the **Message from webpage** dialog box, click **OK**.
19. On the **Shopping Cart** page, click **Place Order**.
20. In the **Message from webpage** dialog box, click **OK**.
21. On the **Orders** page, on the second order, click **Details**.
22. On the **Order Details** page, click the **Backorder Reference** link.
23. Close the web application.
24. In Internet Explorer, in the **SQL API** pane, on the **Query 2** tab, edit the query text so that it reads:

SELECT \* FROM c WHERE c.doctype = "ShoppingCartOrder" AND c.partitionkey = "20"

1. Click **Execute Query**. Two results should be returned.
2. Edit the query text so that it reads:

SELECT \* FROM c WHERE c.doctype = "ProductBackorder"

1. Click **Execute Query**. One result should be returned for **customerid** 20.

**Exercise 4: Maintaining Data and Auditing Orders and Backorders**

**Task 1: Create a Pre-Trigger that Completes the Details of an Order**

1. In Internet Explorer, in the Azure Portal, on the **20777-sql-<*your name>-<the day*>** blade, click **Data Explorer**.
2. In the **SQL API** pane, under **Adventure-Works**, right-click **Data**, and then click **New Trigger**.
3. On the **New Trigger 1** tab, in the **Trigger Id** box, type **CompleteOrderDetails**.
4. In the **Trigger Type** list, click **Pre**.
5. In the **Trigger Operation** list, click **All**.
6. In the **Trigger Body** box, edit the existing code so that it reads:
7. function trigger(){
8. var context = getContext();
9. var request = context.getRequest();
10. var collection = context.getCollection();
11. var documentBeingUpdated = request.getBody();
12. // If the operation being performed is not an upsert, then ignore this trigger
13. if (request.getOperationType() != "Upsert") {
14. return;
15. }
16. // Verify the type of the document
17. if (!("doctype" in documentBeingUpdated) && documentBeingUpdated.doctype != "ShoppingCartOrder") {
18. throw new Error("Wrong type of document");
19. }
20. // Accumulator for counting the number of individual items in the order
21. var numberOfItemsInOrder = 0;
22. // Accumulator for counting the total cost of the order (before any customer discount)
23. var costOfItemsInOrder = 0;
24. // Iterate through the order items in the shopping cart
25. documentBeingUpdated.orderitems.forEach(function (orderitem) {
26. // Calculate the total number of items in the order, and the cost of these items
27. numberOfItemsInOrder += orderitem.numberincartorordered;
28. costOfItemsInOrder += orderitem.lineitemtotalcost;
29. });
30. // Find the discountrate for the customer
31. var isAccepted = collection.queryDocuments(collection.getSelfLink(), "SELECT \* FROM c WHERE c.doctype = 'Customer' AND c.partitionkey = '" + documentBeingUpdated.partitionkey + "'", {}, function(err, customerDocs, options){
32. // Report any errors
33. if (err) {
34. throw new Error(err);
35. }
36. // If the specified customer was not found in the collection throw an error with a suitable message
37. if (customerDocs.length < 1) {
38. throw new Error("Customer " + documentBeingUpdated.partitionkey + " not found");
39. }
40. // Extract the discount rate from the customer document
41. var discountrate = customerDocs[0].discountrate;
42. // Apply the discount to the order
43. documentBeingUpdated.numberofitems = numberOfItemsInOrder;
44. documentBeingUpdated.itemscost = costOfItemsInOrder;
45. documentBeingUpdated.customerdiscountrate = discountrate;
46. documentBeingUpdated.totalcost = documentBeingUpdated.itemscost \* (100 - discountrate) / 100;
47. // Set the order status, date placed, and date last updated
48. documentBeingUpdated.orderstatus = "In progress";
49. documentBeingUpdated.dateplaced = getCurrentTimeInNetTicks();
50. documentBeingUpdated.lastupdated = getCurrentTimeInNetTicks();

});

1. Then type the following code:
2. function getCurrentTimeInNetTicks() {
3. // Get the current time
4. var now = new Date().getTime();
5. // Adjust for 1st Jan 1970
6. const ticksBetweenYear1And1970 = 62135596800000;
7. now += ticksBetweenYear1And1970
8. // Convert the value in now from ms to 100 ns
9. now \*= 10000;
10. // Return the result
11. return now;
12. }
13. // If the trigger is out of runtime, throw an error
14. if (!isAccepted) {
15. throw new Error('Unable to get customer details');
16. }
17. // Save the changes made to the order
18. request.setBody(documentBeingUpdated);

}

1. On the **New Trigger 1** tab, click **Save**.

**Task 2: Create a Post-trigger that Audits Changes Made to Orders and Backorders**

1. In the **SQL API** pane, under **Adventure-Works**, right-click **Data**, and then click **New Trigger**.
2. On the **New Trigger 2** tab, in the **Trigger Id** box, type **AuditOrder**.
3. In the **Trigger Type** list, click **Post**.
4. In the **Trigger Operation** list, click **All**.
5. In the **Trigger Body** box, edit the existing code so that it reads:

function trigger(){

var context = getContext();

var response = context.getResponse();

var collection = context.getCollection();

var documentBeingUpdated = response.getBody();

// Verify the type of the document

if (!("doctype" in documentBeingUpdated) && documentBeingUpdated.doctype != "ShoppingCartOrder" && documentBeingUpdated.docType != "ProductBackorder") {

throw new Error("Wrong type of document");

}

// Create a document with the audit details

var auditDocument = {

partitionkey: documentBeingUpdated.partitionkey,

doctype: "AuditDocument",

doctypebeingaudited: documentBeingUpdated.doctype,

documentbeingaudited: documentBeingUpdated.id,

datechanged: getCurrentTimeInNetTicks(),

statuschangedto: documentBeingUpdated.doctype == "ShoppingCartOrder" ? documentBeingUpdated.orderstatus : documentBeingUpdated.backorderstatus,

customerid: documentBeingUpdated.doctype == "ShoppingCartOrder" ? documentBeingUpdated.partitionkey : documentBeingUpdated.customerid

};

// Add the audit document to the database

var isAccepted = collection.createDocument(collection.getSelfLink(), auditDocument);

// If the trigger is out of runtime, throw an error

if (!isAccepted) {

throw new Error('Unable to create audit record');

}

function getCurrentTimeInNetTicks() {

// Get the current time

var now = new Date().getTime();

// Adjust for 1st Jan 1970

const ticksBetweenYear1And1970 = 62135596800000;

now += ticksBetweenYear1And1970

// Convert the value in now from ms to 100 ns

now \*= 10000;

// Return the result

return now;

}

}

1. On the **New Trigger 2** tab, click **Save**.

**Task 3: Add Trigger Functionality to the Repository**

1. Close any open instances of Visual Studio.
2. On the Start menu, click **Visual Studio 2017**.
3. On the **File** menu, point to **Open**, and then click **Project/Solution**.
4. In the **Open Project** dialog box, in the **File name** box, type **E:\Labfiles\Lab03\Starter\Exercise04\Adventure-Works\Adventure-Works.sln**, and then click **Open**.
5. In the **Security Warning for Adventure-Works** dialog box, clear the **Ask me for every project in this solution** check box, click **OK**.
6. In Solution Explorer, double-click **Web.config**.
7. In App.config, in the **Value** attribute of the **EndpointUrl** key, paste the **URI** value you noted earlier, replacing the text **<URL>**.
8. In the **Value** attribute of the **PrimaryKey** key, paste the **PRIMARY KEY** value you noted earlier, replacing the text **<PRIMARY KEY>**.
9. In Solution Explorer, double-click **Repository.cs**.
10. In Repository.cs, locate the **UpsertItemAsync** method, after the comment **// TODO: Set the TTL of the document**, type the following code:

item.TimeToLive = ttl;

1. After the comment **// TODO: Specify the triggers to be invoked (if any)**, type the following code:
2. var options = new RequestOptions
3. {
4. PreTriggerInclude = preTriggers,
5. PostTriggerInclude = postTriggers

};

1. After the comment **// TODO: Add/replace the document in the collection and invoke the triggers**, replace:

var response = await client.UpsertDocumentAsync(collectionUri, item);

with:

var response = await client.UpsertDocumentAsync(collectionUri, item, options);

**Task 4: Invoke the Triggers**

1. In Solution Explorer, expand **Controllers**, and then double-click **ShoppingCartController.cs**.
2. In ShoppingCartController.cs, after the comment **// Save the order to the database** (line 274), replace:

if (await Repository<ShoppingCartOrder>.UpsertItemAsync(shoppingCart))

with:

if (await Repository<ShoppingCartOrder>.UpsertItemAsync(shoppingCart, preTriggers: new List<string> { "CompleteOrderDetails" }))

1. In Solution Explorer, under **Controllers**, double-click **OrdersController.cs**.
2. In OrdersController.cs, in the **CancelOrderAsync** method, after the comment **// TODO: Save the order, and fire AuditOrder post-trigger** (line 116) , replace:

if (1==0)

with:

if (await (Repository<ShoppingCartOrder>.UpsertItemAsync(orderToCancel, postTriggers: new List<string> { "AuditOrder" })))

1. Still in the **CancelOrderAsync** method, after the comment **// TODO: Save the backorder, and fire the AuditOrder post-trigger**, type the following code:
2. if (await (Repository<ProductBackorder>.UpsertItemAsync(backorderToCancel, postTriggers: new List<string> { "AuditOrder" })))
3. {
4. message.Append($"\nBackorder {orderItem.BackorderReference} cancelled");

}

1. Scroll down to the **CancelBackorderAsync** method at the end of the class. In this method, after the comment **// TODO: Save the backorder, and fire the AuditOrder post-trigger** (line 219), type the following code:
2. if (await (Repository<ProductBackorder>.UpsertItemAsync(backorderToCancel, postTriggers: new List<string> { "AuditOrder" })))
3. {
4. TempData["itemMessage"] = $"Backorder {backorderID} cancelled";

}

1. Press F5 to run and start the application.
2. On the **Adventure-Works Product Catalog** page, in the **Categories** list , click **Clothing**, and then click **Search By Category**.
3. On the first row of search results, click **Details**.
4. On the **Product Details** page, click **Add to Shopping Cart**.
5. On the **Login** page, in the **Customer ID** box, type **30**, and then click **Login**.
6. In the **Message from webpage** dialog box, click **OK**.
7. On the **Shopping Cart** page, click **Place Order**.
8. In the **Message from webpage** dialog box, click **OK**.
9. On the **Order Details** page, click the **Adventure-Works Product Catalog** link.
10. On the **Adventure-Works Product Catalog** page, in the **Categories** list , click **Clothing**, and then click **Search By Category**.
11. On the first row of search results, click **Details**.
12. On the **Product Details** page, in the **Number Required** box, type **5000**, and then click **Add to Shopping Cart**.
13. In the **Message from webpage** dialog box, click **OK**.
14. On the **Shopping Cart** page, click **Place Order**.
15. In the **Message from webpage** dialog box, click **OK**.
16. On the **Orders** page, on the second order, click **Details**.
17. On the **Order Details** page, click the **Backorder Reference** link.
18. Close the web application, and then close Visual Studio.
19. In Internet Explorer, in the **SQL API** pane, on the **Query 2** tab, edit the query text so that it reads:

SELECT \* FROM c WHERE c.doctype = "ShoppingCartOrder" AND c.partitionkey = "30"

1. Click **Execute Query**. Two results should be returned.
2. Edit the query text so that it reads:

SELECT \* FROM c WHERE c.doctype = "ProductBackorder"

1. Click **Execute Query**. Three results should be returned - two from Exercise 3, the other from Exercise 4.

**Task 5: Lab Close Down**

To reduce your costs delete the resource group containing your Cosmos DB databases:

1. In Internet Explorer, in the Azure Portal, click **Resource groups**.
2. Right-click **20777\_Mod03**, and then click **Delete resource group**.
3. On the **Are you sure you want to delete "20777\_Mod03"?** blade, in the **TYPE THE RESOURCE GROUP NAME:** box, type **20777\_Mod03**, and then click **Delete**.
4. When you have finished, close Internet Explorer.