

Entity Framework and SQL Azure

Julie Lerman

<http://thedatafarm.com>

April 2011

As part of Microsoft's Azure platform, SQL Azure is your relational database in the cloud. In fact, SQL Azure is very close to being SQL Server in the cloud except for features and functions that don't apply to SQL Azure. This includes things like data file placement, server configuration and backups – functionality that is automatically handled for you by the service.

However, one of the very important features that it does have in common is integration with Visual Studio 2010. You can use all of the same designer tools in Visual Studio to work with SQL Azure as you can to work with SQL Server. The Server Explorer allows you to view (but not edit) database objects. The Visual Studio Entity Data Model Designer is fully functional and you can use it to create models against SQL Azure for working with Entity Framework.

In this walkthrough you'll learn how to create an Entity Data Model directly from a SQL Azure database as well as create a SQL Azure database using the EDM Designer's model-first feature. You'll also learn about where you will pay the highest price for latency when your on-site applications are connecting to SQL Azure and how you can leverage Entity Framework's querying capabilities to reduce that latency.

I'll use an existing SQL database, AdventureWorksLT2008R2, which you can get from [CodePlex Microsoft Community Samples](#) which I've migrated to SQL Azure and named AdventureWorksLT. Because of the differences between SQL Azure and SQL Server, you will need a specially written script to create the database rather than using the same one that you would use to create the database in SQL Server. I used the SQL Azure Migration Wizard from CodePlex.com. Additionally, the November 2010 MSDN Magazine article, [Getting Started with SQL Azure Development](#) should give you the information you need to get started with SQL Azure and to migrate an on-premises SQL Server database to the cloud.

Database First Modeling with SQL Azure

Creating a database first model against a SQL Azure database is no different than creating it against a database on your network. You simply need to point to the cloud database instead of something local.

It is a good practice to create Entity Data Models in their own Class Library project so that you can reuse them in different applications so that's what you'll do here.

1. Create a new Class Library project in Visual Studio.
2. Name the project AdventureWorksModel.
3. Delete the Class1 class file as you won't be using it.
4. Add New Item to the project and select ADO.NET Entity Data Model
5. Change its item name to AdventureWorksLT.edmx and then click Add.
6. In the Choose Model Contents page of the Entity Data Model Wizard, select Generate from database and click Next.
7. In the Choose Your Data Connection page of the wizard, click the New Connection button to open up the Visual Studio Connection Properties wizard.
8. Type the name of your SQL Azure server into the Server name text box. For example, myserver.database.windows.net.
9. Select Use SQL Authentication and enter your SQL Azure User name and Password. User name should follow the pattern of username@myserver
10. If you entered the authentication details correctly, you should be able to drop down the database list under Select or enter a database name and see your installed AdventureWorks LT database.
11. Choose AdventureWorksLT then click OK to complete the Connection Wizard and return to the Entity Data Model wizard which will have the connection pre-selected for you.
12. As this is only sample walkthrough, choose the option to include the sensitive data (e.g., your password) in the connection string.
13. Click the Next button to see the page where you can choose which database objects should be included in the model.
14. Check Tables, then expand the Tables node and uncheck BuildVersion and ErrorLog.
15. Check Views.
16. Check Stored Procedures.
17. Click Finish.

The Entity Data Model Wizard will then create a model based on your selections from your SQL Azure database. Figure 1 shows that model with the entities that were created from the selected database tables. The relationships between the tables as defined by database constraints are reflected by associations between the entities. Each table's primary key designation has been interpreted as an Entity Key in the relevant entity. Two entities in this model were created from the selected database views, vProductAndDescription and vGetAllCategories.



Figure 1: Model of the AdventureWorksLT2008 Database

The Entity Data Model Wizard was able to perform all of the same tasks when working directly with the SQL Azure database as it would perform against a local database instance.

You can also create any model customizations that you need whether that is renaming entities or properties, defining inheritance hierarchies, splitting entities, mapping or importing stored procedures.

This particular database does have a few stored procedures for logging and printing errors. You can bring those into your conceptual model using the steps laid out in the MSDN document [How to: Import a Stored Procedure \(Entity Data Model Tools\)](#).

Creating a SQL Azure Database with Model-First Development

When beginning a new development project you don't always have an existing database to work with. The Entity Model Designer supports model-first design, allowing you to create your Entity Data Model in the designer and then generate a database based on the model. This feature can be used with SQL Azure.

To do model-first design, you can create a new ADO.NET Entity Data Model item, but on the first page of the wizard, rather than choosing *Generate from database* as you did in the previous walkthrough, you must select *Empty model*. This will present you with a blank EDM Designer and the tools you need to create entities, their properties and associations as well as build inheritance hierarchies and perform other advanced model mappings. You can learn about model-first design and the tools of the Entity Data Model Designer in the MSDN Library topic, ADO.NET Entity Data Model Designer, at <http://msdn.microsoft.com/en-us/library/cc716685.aspx>.

Figure 2 displays a model created in the designer to keep track of historical information about cryptanalysis.

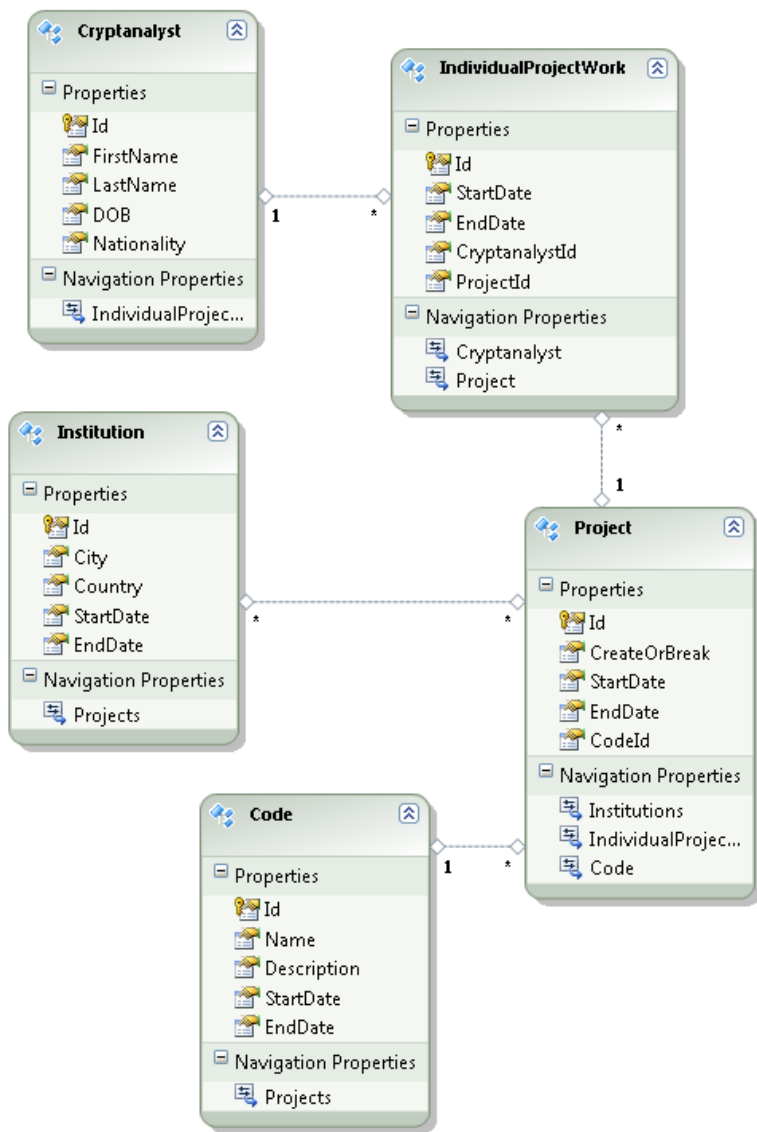


Figure 2: A model build in the Entity Data Model designer

Once you have defined the model, you can generate a database and its schema through its context menu. To do this:

1. Right click on the designer surface.
2. Click *Generate Database from Model* from the model's context menu.
3. Because you will be creating a new database, you will also need a new connection.
4. In the Generate Database Wizard screen, click the New Connection button.
5. Fill out the SQL Azure server information along with your user name and password.
6. Type in the name for the new database, e.g., Cryptanalysis, as shown in Figure 3.
7. Click OK.

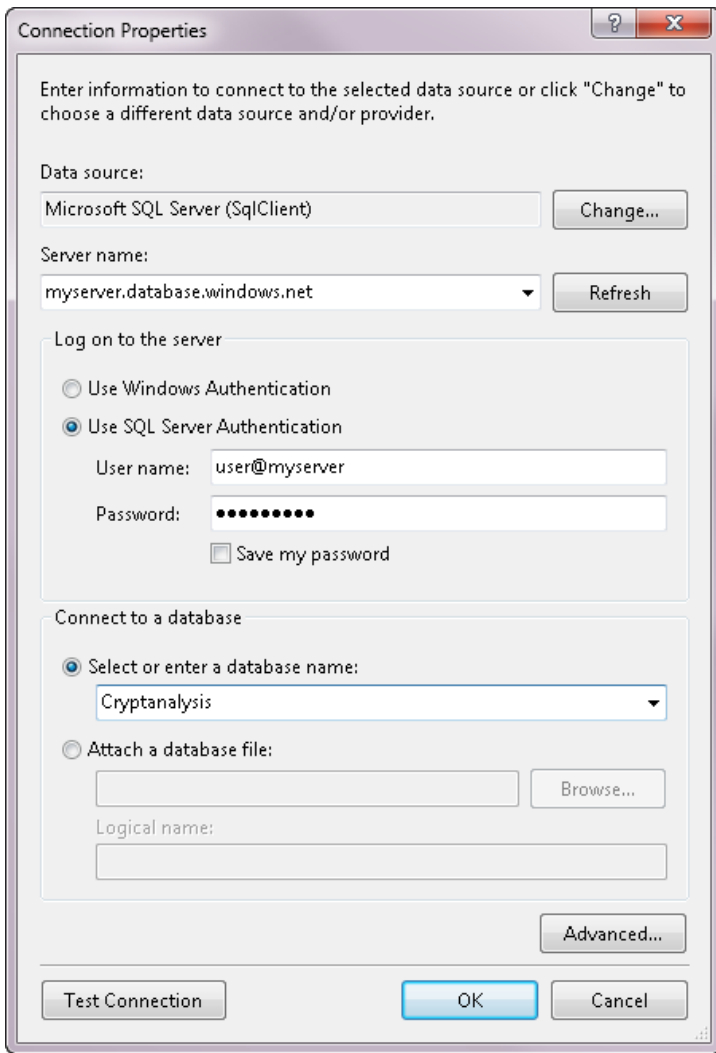


Figure 3: Connecting to the SQL Azure server to create a new database

8. You will get a dialog box stating that the database does not yet exist and should Visual Studio attempt to create it. Answer Yes to this question.

Browsing to your SQL Azure instance in the cloud, you can see that the new database is there, but it does not yet have its tables and other objects.

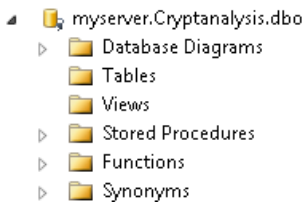


Figure 4: Empty database created by the Connection Properties wizard

If you were using the model-first feature with a local database, you would have seen exactly the same behavior. Visual Studio sees no difference between SQL Azure in the cloud and a local database.

When the wizard has finished creating the database, the Choose your connection window will reappear with the new database selected and the connection string displayed in the center of the window. This screen will be waiting for you to select an option to exclude or include the sensitive data (your database password) in the connection string.

Once you have made this selection you will be able to click the Next button which will result in the Data Definition Language (DDL) script displayed. At the top of the script you'll see a note that says "Entity Designer DDL Script for SQL Server 2005, 2008, and Azure". Scrolling through the window, you will see T-SQL script to create tables and the other necessary objects of the database. The wizard does not execute the script on your behalf. You'll need to do this yourself.

Click Finish to close the DDL window.

Notice that the model project now contains a two new files. An app.config and a sql file.

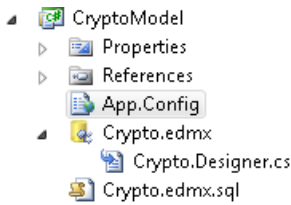


Figure 5: New config and sql files created by the wizard

Before running the SQL Script it is important to note one of the limitations of SQL Azure. It does not support the SQL USE statement. This is because each database on a SQL Azure server is typically located on a separate physical server and therefore switching database context after a connection has been made to a different physical server is not supported.

This means that not only do you need to remove the third line of code in the script (USE [Cryptanalysis]), but that you must force Visual Studio to point to the Cryptanalysis database before running the script. You can do this from the SQL Editor window.

1. Right click in the background of the SQL script in the editor to open its context menu.
2. In the context menu, choose Connection.

If you are already connected to another database, the Connect command will be disabled. In this case, you will need to first choose Disconnect and then open the context menu again.

3. Choose Connect from the Connection option of the context menu.
4. In the Connect to Database Engine window, fill out the relevant database connection information and then click the Options button.
5. In the Connection Properties window, type the name of the database, Cryptanalysis, into the Connect to database option.
6. Click Connect.

If the connection was made properly, you should see the Cryptanalysis database listed in the Transact-SQL Editor menu as shown in Figure 5.

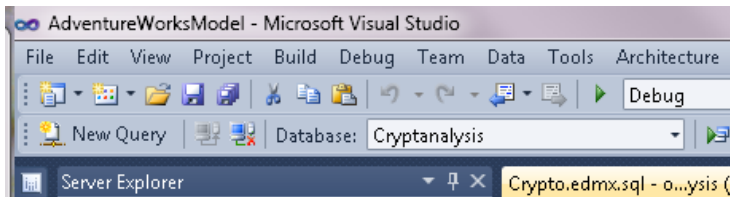


Figure 6

7. Delete the Use [Cryptanalysis] code line from the SQL script if you have not already.
8. Right click the SQL Editor window to re-open the context menu and choose Execute SQL from the context menu.

Alternatively, you can use the Connect, Disconnect and Execute SQL icons on the SQL Editor menu to perform these tasks.

When the query is finished executing, refresh the view of the database in Server Explorer to see all of the tables that were created, including the join table to represent the many-to-many relationship between Project and Institution. All of the primary keys as well as primary foreign key constraints defined in the model are also reflect in the database; just as they would be had you run the script against a local database.

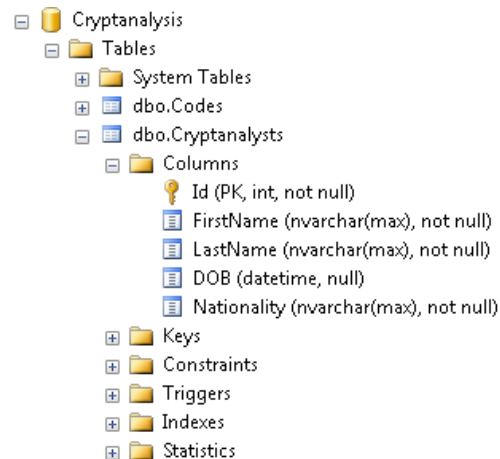


Figure 7: Tables created in SQL Azure

Using the Model and SQL Azure data in an Application

With the model created, you can begin coding your application just as you would if the model had been created locally. As an example, following is a set of data access code for retrieving a list of Customers, retrieving a single customer and for updating a customer.

```

1. public static List<Customer> GetPagedCustomers(int skip, int take)
2.     {
3.         using (var context = new AdventureWorksLTEntities())
4.         {
5.             var query = from c in context.Customers

```

```

6.         orderby c.LastName + c.FirstName
7.         select c;
8.     return query.Skip(skip).Take(take).ToList();
9.     }
10. }
11. public static Customer GetCustomerById(int id)
12. {
13.     using (var context = new AdventureWorksLTEntities())
14.     {
15.         var query = from c in context.Customers
16.                     where c.CustomerID == id
17.                     select c;
18.         return query.Single();
19.     }
20. }
21. public static void UpdateCustomer(Customer customer)
22. {
23.     using (var context = new AdventureWorksLTEntities())
24.     {
25.         context.Customers.Attach(customer);
26.         context.ObjectStateManager.ChangeObjectState(customer, EntityState.Modified);
27.         context.SaveChanges();
28.     }
29. }

```

These methods were built for use in an ASP.NET MVC application although they are flexible enough for any layered application. Because these methods are designed to be used in a disconnected application, the context is instantiated and disposed in each method. This is not particular to SQL Azure but is a common pattern for Entity Framework.

SQL Azure and a Long-RunningObjectContext

If you were writing a client-side application, such as a Windows Forms or WPF application, you should consider using a long-running context which can track changes to entities.

In the MVC application, the applications' Customer controller calls the methods and then passes results to a series of Customer views. When a Customer has been edited by a user, that entity is then sent to the UpdateCustomer method and updated using a new context. A data access class with a long-running context would declare the context in the class declarations and then subsequently use the same context in its methods.

Following is partial example of a data access class that would be more suitable for a client side application. It contains a method to return all Customers.

```

1. public class DataAccess
2. {
3.     private AdventureWorksLTEntities _context;
4.     public DataAccess()
5.     {
6.         _context = new AdventureWorksLTEntities();
7.     }
8.     public static List<Customer> GetCustomers()
9.     {
10.         var query = from c in context.Customers
11.                    orderby c.LastName + c.FirstName
12.                    select c;
13.         return query.ToList();
14.     }
15. }

```

A common concern with long-running contexts, especially those that interact with SQL Azure, is that they might hold the connection open for a prolonged period of time. This is not the case. By default, the Entity Framework ObjectContext closes its database connection as soon as it is finished capturing the results of the query.

In the GetCustomers method, as soon as all of the results are returned from the query.ToList() call, the ObjectContext will close the DbConnection that was used to execute the query in the database.

Testing Against Local Database

While building your applications and verifying your application, whether you run the actual executable or use automated tests, the connection

string will force all of your data to come from and be persisted to your cloud database.

While it is important to be sure that you are truly interacting with your SQL Azure database, it is advisable to do most of your development testing against a local copy of the database. This is because of the natural latency that you will experience when interacting with a database in the cloud.

Switching to the local database during development means nothing more than changing the connection string in your executing application. If you have your model in its own project, the connection string in that project's app.config file is used by the Entity Data Model Designer any time it needs to interact with the database. It is the connection string in your startup project, for example, the MVC application that is consuming my model, which is in control of the runtime database connection.

When I originally created the MVC application that uses the model, I copied the connection string from the model project into the MVC application's web.config file.

```
1. <add name="AdventureWorksLTEntities"
2.     connectionString="metadata=res://*/AdventureWorksLT.csdl|
3.                       res://*/AdventureWorksLT.ssdl|
4.                       res://*/AdventureWorksLT.msl;
5.                       provider=System.Data.SqlClient;
6.     provider connection string=
7.         'Data Source=mserver.database.windows.net;
8.         Initial Catalog=AdventureWorksLT;
9.         User ID=user@ovhb5xakjw;Password=mypassword;
10.        MultipleActiveResultSets=False'
11.     providerName="System.Data.EntityClient"
12. />
```

To point to the local instance of the database, simply replace the provider connection string parameter of the connectionString with a connection that points to the local database such as this one that points to a local SQL Server instance.

```
'Data Source=.;Initial Catalog=AdventureWorksLT2008R2;
Integrated Security=True;MultipleActiveResultSets=False'
```

Your Entity Framework code will continue to work. The only difference is that queries and updates to the database will execute more quickly. As you are designing and testing your application repeatedly, you will certainly appreciate the faster database interactions.

Modeling Against a Local Database then Switching to SQL Azure

Just as you can build a model against a SQL Azure database and switch the connection string to point to a local database, the reverse is true as well. You can create a database first model from a local database and change the connection string at a later time to run the application against a duplicate SQL Azure database. You could also use model-first and the resulting DDL to build the schema of a local database at the same time as building the schema of a SQL Azure database. If you are starting with a local database, you should be aware in advance what SQL Server features are not supported in SQL Azure. That way you will not encounter any surprises if you do all of your work against a local database and create the cloud database when you are far along in the development process. The MSDN Library document, Guidelines and Limitations (SQL Azure Database) at <http://msdn.microsoft.com/en-us/library/ff394102.aspx> is a great starting point to discover any difference you need to be prepared for.

Reducing Latency Between On-Premises Apps and the Cloud

Although it is beneficial for your development process to work with the local database, you do not want to become complacent about the impact that latency will have on your application when it goes into production. Not only should you profile the performance while working against the local database, it is critical that you do this again with your connection string is pointing to the SQL Azure database.

While the latency of hitting the far away database is unavoidable, it is interesting to note that the latency impacts Entity Framework in different ways and that you can tune your application to achieve the best possible performance and minimize the intrusion of latency.

In the MVC application, some of the worst latency problems will not even require a profiling tool. Your eyes will be enough to see the latency while your application is running.

The using pattern used in the methods to ensure that a context is short-lived will help you to avoid one of the biggest victims of latency – lazy loading. Lazy loading is used in an application when the query doesn't explicitly request related data but the code attempts to interact with it. By default, Entity Framework will use lazy loading to retrieve related data on demand without an explicitly query. This feature is a great benefit in many scenarios; however it is a pattern that some developers will use without even being aware that they are causing it to be called into action.

A commonly used pattern, which is also a software anti-pattern, is to query for a set of data and then expect the UI to load that data along with some related information. For example, your code returns a list of customers, but the UI wants to display customers along with their order information. As the UI is rendering its view, it will cause EF to return to the database to get the related data. Not just once, but one trip for each customer. In other words as it is rendering the first customer and then wants the orders for the customer, the Entity Framework to execute a query on the database for that customer's orders. Then as the view renders the next customer, EF will execute another database query to get that customer's orders.

In a web page such as that shown in Figure 8, eleven trips would be made to the database. The first to get the customers and then one to get the orders for Action Bicycle Specialists, another to get the orders for Aerobic Exercise Company and so forth. When the database is local, those extra trips may not be noticeable. But when the database is SQL Azure, those ten extra trips may be very noticeable.

AdventureWorks MVC

```
Action Bicycle Specialists
*6/1/2008: $108,561.83

Aerobic Exercise Company
*6/1/2008: $2,137.23

Bulk Discount Store
*6/1/2008: $88,812.86

Central Bicycle Specialists
*6/1/2008: $38.95

Channel Outlet
*6/1/2008: $550.39

Closest Bicycle Store
*6/1/2008: $35,775.21

Coalition Bike Company
*6/1/2008: $2,415.67

Discount Tours
*6/1/2008: $2,980.79

Eastside Department Store
*6/1/2008: $83,858.43

Engineered Bike Systems
*6/1/2008: $3,398.17
```

Figure 8: An ASP.NET MVC App displaying Customers with related data

In a well-designed application, enabling the user interface to trigger data access activity is also what's referred to as an anti-pattern. SQL Azure would help to exaggerate the problems of this pattern. If you attempted to render this page using the `GetPagedCustomers` method above, an exception would be thrown because that method follows the pattern of disposing the context immediately. When the view attempts to trigger lazy loading, there is no `ObjectContext` available.

Consider Eager Loading over Lazy Loading

A better solution in scenarios where you know that you want the related data is to use eager loading. In Entity Framework that means using the `Include` method. Here is the query with that modification along with an additional way of ensuring that the query only returns Customers who have orders.

```
1. var query = from c in context.Customers.Include("SalesOrderHeaders")
2.             orderby c.CompanyName,c.LastName + c.FirstName
3.             select c;
4.     return query.Skip(skip).Take(take).ToList();
```

Now the orders will be returned along with the customers and available immediately for the view to display.

The November 2010 Data Points column in MSDN Magazine focuses on the latency that will naturally occur when your on-premises applications interact with a SQL Azure database. It highlights problem areas and proposes solutions. In addition to eager loading, the article discusses the benefits of using projections to fine tune the eager loaded data even further.

Using Projections to Tune Queries

If you were really only interested in returning orders with a minimum subtotal, perhaps \$10,000, you can use a projection to ensure that only those orders are retrieved from the database. This is not something you can achieve with the `Include` method and would help to reduce the amount of data being returned from the database, which in turn, will reduce the latency cost of the query.

This will require a bit of refactoring in the data access class because a query projection will, by default, return an anonymous type. But you cannot use an anonymous type outside of the method in which it was created. Therefore, you'll need a new class to support the projected results. Here is the modified method along with the new `CustomerOrders` class that will be returned.

```
1. public class CustomerOrders
2.     {
3.         public Customer Customer { get; set; }
4.         public IEnumerable<SalesOrderHeader> Orders { get; set; }
5.     }
6.     public static List<CustomerOrders> GetPagedCustomers(int skip, int take)
7.     {
8.         using (var context = new AdventureWorksLTEntities())
9.         {
10.            var query = from c in context.Customers.Where(c => c.SalesOrderHeaders.Any())
```

```
11.         orderby c.CompanyName , c.LastName + c.FirstName
12.         select new CustomerOrders {Customer = c,
13.             Orders=c.SalesOrderHeaders.Where(o => o.SubTotal >= 10000)};
14.     return query.Skip(skip).Take(take).ToList();
15. }
16. }
```

Windows Azure Apps and SQL Azure

If you plan to host your applications in Windows Azure instead of on-premises servers, you will not experience these latency issues as your application interacts with SQL Azure. Windows Azure and SQL Azure were designed to work together.

You can host web applications as well as services in Windows Azure. Although Microsoft has data centers all over the world for hosting Azure services, your applications and cloud database can be located in the same datacenter for the greatest efficiency.

You can build Windows Azure applications in Visual studio and have them consume a SQL Azure database. At development time, these Azure applications will be hosted on your development machine so you will still want to work with a local database for application testing.

Summary

In many ways, using SQL Azure as the data store for your Entity Framework data access layer is not much different than working against a locally hosted database. You can use the designer tools directly with SQL Azure if you like but you do need to pay attention to latency not only when debugging your application, but when choosing your querying strategies. SQL Azure is still evolving and the SQL Azure team is working to provide the entire stable of SQL Server products in the cloud.

About the Author

Julie Lerman is a Microsoft MVP, .NET mentor and consultant who lives in the hills of Vermont. You can find her presenting on data access and other Microsoft .NET topics at user groups and conferences around the world. Julie blogs at thedatafarm.com/blog and is the author of the highly acclaimed book, "Programming Entity Framework" (O'Reilly Media, 2009). Follow her on Twitter.com: [julielerman](https://twitter.com/julielerman).