|  |
| --- |
|  |
| WCF Data Services |
| LINQ to REST |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

LINQ and WCF Data Services

# Overview

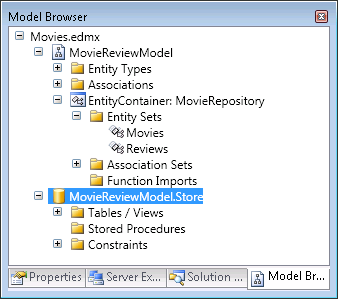
In this lab, we will create a WCF data service using the ADO.NET Entity Framework as a data source. We will also create query interceptors and add service operations to the data service. We will consume the service using a .NET console program.

# Part I – Creating a Data Service

1. This lab requires the moviereviews database. If you’ve already installed the database you can move to the next step. To install the database, execute the SQL commands in the Scripts\moviereviews.sql file where you’ve extracted the Pluralsight labs. You can execute the script file using SQL Management Studio, sqlcmd, or any other SQL tool. If you need help, the file Labs\DatabaseSetup\_Troubleshooting.doc file can walk you through the installation.
2. In Visual Studio, use the File -> Open Web Site command to open the *LINQ\_DataServices\before\****WebSite*** directory. The *LINQ\_DataServices\after* directory contains a completed version of this lab.
3. Double click the Movies.edmx file in the App\_Code folder. This will open the design window for the MovieReviews entity model.

Movies.edmx is a model built with the ADO.NET Entity Framework. The model will not know what database it is associated with, so you might see error message if you try to build the project or manipulate the model. We will associate the model with a local database in the next step.

1. Right-click an area of white space in the designer window and select “*Update Model From Database…*”. This will launch the Update Wizard.
2. In the opening screen of the wizard, select a connection to the moviereviews database you’ve installed. If you do not have an existing connection to the database, click on the *New Connection* button to establish the connection.
3. Once you’ve establish a connection, ensure the “*Save entity connection settings in Web.Config as:*” checkbox is selected and click the *Next* button.
4. On the second screen of the Update Wizard (*Choose Your Database Objects*), click the *Finish* button.
5. Once the wizard is finished, right-click in the white space of the designer window and select “*Properties*”.
6. In the Properties Window, note the *Entity Container Name* (***MovieRepository***). This class is derived from the Entity Framework’s *ObjectContext* class and represents the gateway to the database. Also note the namespace for the class, which should be set to ***MovieReviewModel***. We will need this information to configure our service.
7. Right-click the white space in the designer window again and select “*Model Browser*”. In the Model Browser window, expand the *EntityContainer* node, and then the *Entity Sets* node. Notice our model has two entity sets named ***Movies*** and ***Reviews***. We will need to know these names to configure access rights in our service.



1. Your model is now ready to work with your moviereviews database, so close the Movies.edmx file and save any changes.
2. Right click the web site node in the Solution Explorer window.
3. In the *Add New Item* dialog, select the *WCF Data Service item*. Give the new item a name of *MovieReviews.svc*, then click *Add*.
4. Visual Studio should now present you with the source code for MovieReviews.cs. Take a moment to review the comments in this file.
5. The first TODO comment in this file is telling us to put our data source class name as the generic type parameter for the data service. The class we need to use is the MovieRepository class generated from our entity data model. Add a *using* statement for the MovieReviewModel namespace at the beginning of the file, and add the MovieRepository class as the generic type parameter.

// ...

using MovieReviewModel;

public class MovieReviews : DataService<MovieRepository>

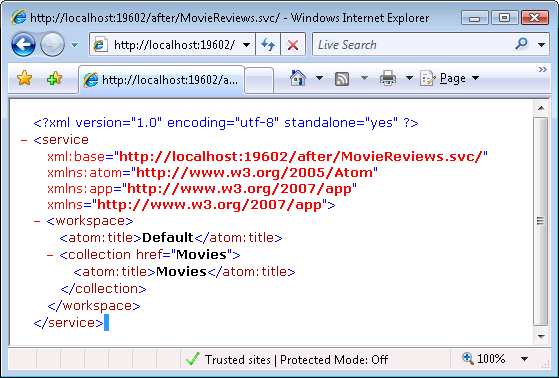
1. Inside the static InitializeService method we need to configure access rules for our entities. Uncomment the line of code with the method call *SetEntitySetAccessRule*. Replace the first parameter (“*MyEntitySet*”) with the string “*Movies*”.

config.SetEntitySetAccessRule("Movies",

EntitySetRights.AllRead);

Remember the Entity Sets we explored earlier in the Model Browser? We have two entity sets to configure – Movies and Reviews. We could allow unlimited access to both entity sets with the following code: config.SetEntitySetAccessRule("\*", EntitySetRights.All); However, we are going to make access more granular in this lab.

1. At this point, we should have a working data service. Right-click MovieReviews.svc and select “*View In Browser*”, saving any changes if prompted. You should see an XML response in the browser like the following:



1. The response is telling us there is a collection of movies available at the relative URL *Movies*. In the address bar of the web browser, add */Movies* to the end of the current URL, and press enter to load the new address.

Note: URLs for an ADO.NET data service are case sensitive. If you use /movies instead of /Movies, you’ll receive an HTTP 404 error.

If you are having other problems with the service, there are two steps to take to see more error information. The first step is to set the EnableVerboseErrors property on the incoming IDataServiceConfiguration parameter.

The second step is to configure WCF to allow exception details to flow into the response using a ServiceBehavior attribute on your service. You’ll want to disable these steps in production code, but the following snippet can help you if you need to debug problems:

**[System.ServiceModel.ServiceBehavior(**

**IncludeExceptionDetailInFaults = true)]**

public class MovieReviews : DataService<MovieRepository>

{

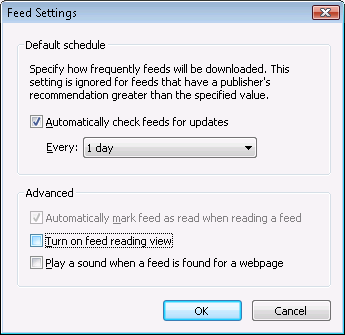
public static void InitializeService(

IDataServiceConfiguration config)

{

**config.UseVerboseErrors = true;**

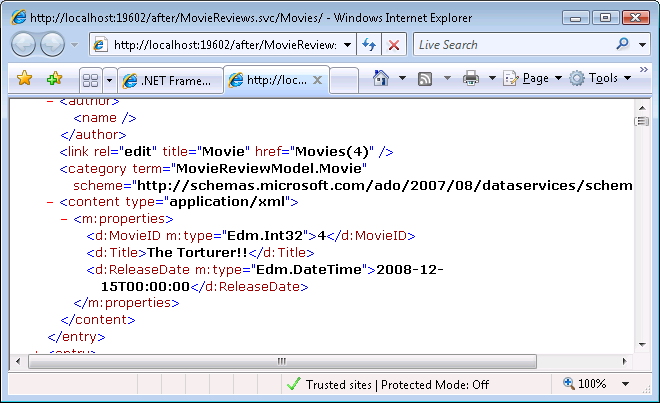
1. Depending on your configuration, Internet Explorer 7 is probably showing you a “feed reader” view of the XML response. If you do not see any XML in the browser, take the following steps:
   1. Go to *Tools* -> *Internet Options*.
   2. Go to the *Content* tab.
   3. In the *Feeds* area (bottom of the dialog), click *Settings*.
   4. Uncheck the “*Turn on feed reading view*” textbox.



* 1. Click *OK*, then copy the URL in Internet Explorer’s address bar.
  2. Open a new instance of Internet Explorer, and paste the address into the address bar. Press Enter to load the address.

Note: you may need to wait some time for Internet Explorer to process the response. The data service will send back XML for every movie in our database. The large amount of data can take some time to download and render in the browser. We will fix this potential problem later. Also note that Internet Explorer will still try to make the XML “look pretty” with some formatting that might obscure the actual results of the service. In these cases, using a tool like HTTP Fiddler (or simply the View -> Source menu command) are the best options.

1. You should be able to see all the Moves represented as XML.



1. Notice the primary key for each movie is included in the response, and is part of an *href* attribute in a *<link>* tag that describes the movie.
2. Navigate to a specific movie by including a movie ID in the URL. The movie ID must be enclosed by parentheses. For example, an URL that ends like the following will retrieve the XML for the movie with a primary key of 100:

MovieReviews.svc/Movies(100)/

What we’ve done to this point is test our data service using “GET” operations from the browser. The data service looks at the incoming GET request for MovieReviews.svc and inspects the URL to see what it should be retrieving.

1. Ideally, we should be able to see the reviews for a movie. Try using the following string at the end of the URL:

MovieReviews.svc/Movies(100)/Reviews/

1. The service should return a 404 response, because we have not given any access rights to for Review entities. Close the browser.
2. Return to Visual Studio and the MovieReviews.cs file in App\_Code. After the line of code that grants access rights to Movies, add a new access rule that grants all rights for the Reviews entity set.

config.SetEntitySetAccessRule("Movies",

EntitySetRights.AllRead);

**config.SetEntitySetAccessRule("Reviews",**

**EntitySetRights.All);**

1. Let’s make one more change to limit the number of entities sent in the response. Remember when we requested all of the movies from the database using the browser (MovieReviews.svc/Movies)? We wouldn’t want someone accidently issuing a query that overloads our network and database, so add a line of code beneath the access rules that sets the MaxResultsPerCollection property of the config object to 20.

config.MaxResultsPerCollection = 20;

1. Save the file, then right-click the *MovieReviews.svc* file in Solution Explorer and select *View in Browser*. You should now see Reviews listed as an available collection.
2. Add the following to the end of the address in the browser:

/Movies

1. Notice the browser will respond much faster than it did previously, we have limited the results to 20 movies. Now use the following URL:

/Movies(100)/Reviews

1. You should now see the reviews listed for the movie with an ID of 100. We can also look at all Reviews (although the result set is limited to 20):

MovieReviews.svc/Reviews/

1. In the next section, we will look at using queries to restrict our results set.

# Part II - Querying With The Browser

1. Using “$ options” in the URL, we can send query operators and special processing instructions to the data service. For instance, try the following URL, which will display metadata about our data service. We will use metadata later when we build a .NET client.

MovieReviews.svc/$metadata

1. We can also ask the data service to filter the results before it constructs a response. Try the following URL to find the Star Wars movie. The URL uses a $filter operator in the query string.

MovieReviews.svc/Movies/?$filter=Title eq 'Star Wars'

1. There is also an $orderby operator. Try to create an URL that will order the movies by their Title property.

Movies/?$orderby=Title

1. We can concatenate operators using the standard & character in the URL. Try ordering the movies by their Title property, and then selecting the top 10 (using the $top operator).

MovieReviews.svc/Movies/?$orderby=Title&$top=10

1. There is also a $skip operator, which is useful for paging operations. Try creating a query that will retrieve the 100th to 109th movie after ordering the movies by their Title.

MovieReviews.svc/Movies/?$orderby=Title&$skip=99&$top=10

1. Let’s concentrate on a specific movie. Create an URL that will pull the movie with an ID of 100.

MovieReviews.svc/Movies(100)/

1. Now create an URL that will retrieve the reviews for the movie with an ID of 100.

MovieReviews.svc/Movies(100)/Reviews/

1. An obvious question at this point might be: how can we retrieve the data for the movie with an ID of 100 **and** include all the movie’s reviews? This is the job of the $expand operator. Try using the following URL:

MovieReviews.svc/Movies(100)/?$expand=Reviews

1. Notice the XML response includes not only the movie details, but all of the reviews for the movie.
2. We can also request specific properties of a movie. Try the following URL.

MovieReviews.svc/Movies(100)/Title/

1. We have a response that includes only the title of the movie with an ID of 100. The response still includes XML. To remove the XML, use the $value directive (note the directive does directly in the URL, not the query string).

MovieReviews.svc/Movies(100)/Title/$value

1. Some queries are difficult to express in an URL (other queries are impossible to express in the URL with the limited number of operators we have available). In the next section, we will create a service operation and use a query interceptor to improve our query capabilities.

# Part III – Query Operations and Interceptors

1. Return to Visual Studio and open the MovieReviews.cs file in App\_Code.
2. Let’s define a service operation that will return the top 10 movies for a given year, based on the movie’s review ratings. First, we’ll need to add a public method to our data service that returns an IQueryable of Movie objects.

public IQueryable<Movie> GetTopTenMovies(int year)

{

}

1. Inside the method, write a LINQ query against the CurrentDataSource property of the service that will return the top 10 movies from the specified year, and order the movies by their average rating in descending order.

var query =

from m in CurrentDataSource.Movies

where m.ReleaseDate.Year == year

orderby m.Reviews.Average(r => r.Rating) descending

select m;

return query.Take(10);

1. We need two additional steps to allow web access this operation. First, decorate the new method with a WCF [WebGet] attribute.

[WebGet]

public IQueryable<Movie> GetTopTenMovies(int year)

{

// ...

}

1. Secondly, inside the static InitializeService method of the service, add an access rule for the operation.

config.SetServiceOperationAccessRule("GetTopTenMovies",

ServiceOperationRights.AllRead);

1. Now launch a browser and view the operation results with the following URL.

MovieReviews.svc/GetTopTenMovies?year=2006

1. Notice the *year* parameter is passed in the query string by name. Also, since the method returns an IQueryable sequence of objects, we can continue composing operating into the query using $ operators. Try the following URL.

MovieReviews.svc/GetTopTenMovies?year=2006&$expand=Reviews&$orderby=Title

1. The above URL reorders the movies returned by the operation and expands each movie’s associated review into the response.
2. Now return to the MovieReviews.cs file in App\_Code. We are going to implement a query interceptor.

Service operations are chunks of logic and queries that a client can invoke directly. Query interceptors are implicit operations that we can apply to requests. For example, in an authenticated data service we might use an interceptor to inspect an incoming user’s identity and restrict the entities they see in the results.

1. Add a *using* statement for the *System.Linq.Expressions* namespace.

A query interceptor is a public method that returns an object of type System.Linq.Expressions.Expression<T>, where T represents a predicate for a specific entity type in our service. The data service will use all interceptors for a given entity type by grafting the expressions into the query used to retrieve entities.

1. Create a public method that returns Expression<Func<Movie, bool>>. Name the method MovieYearInterceptor.

public Expression<Func<Movie, bool>> MovieYearInteceptor()

{

}

1. Inside the method, return a predicate expression that will filter out movies released before the year 2007 (you can build the expression using lambda syntax).

public Expression<Func<Movie, bool>> MovieYearInteceptor()

{

**return m => m.ReleaseDate.Year >= 2007;**

}

1. For the data service to recognize the query interceptor, we need to decorate the method with an attribute.

[QueryInterceptor("Movies")]

public Expression<Func<Movie, bool>> MovieYearInteceptor()

…

1. Save your changes and try to query the data service for movies released before the year 2007. The URL for the query will look like the following.

MovieReviews.svc/Movies/?$filter=year(ReleaseDate) lt 2007

1. If the query operator is performing correctly, the above query should not return any data. Try changing the *lt* operator in the URL to a *gt* operator to see movies returned by the service.

MovieReviews.svc/Movies/?$filter=year(ReleaseDate) gt 2007

1. You might also want to try the service operation we created earlier.

MovieReviews.svc/GetTopTenMovies?year=2006

1. The above query ***will*** return data. Query interceptors do not apply to service operations.

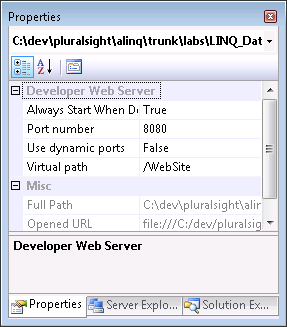
# Part IV – Creating a .NET Client

1. Go to the *File* menu in Visual Studio and select *Add -> New Project*.
2. Under the *Windows* project types, select the *Console Application* project template. Give the project the name DataServiceClient. Set the location of the project to the *before* directory of this lab.

The web site project we’ve been using will run using the WebDev development web server on a random port. Random ports can create some difficulties when used with service end points, so the following steps will switch the project to a fixed port.

1. Click on the web site in the Solution Explorer window and press F4.
2. You should now see the Properties Window for the lab’s web site. Set the “Use Dynamic Ports” property to *false*, and close the Property Windows.
3. Click on the web site in the Solution Explorer window and press F4.
4. Set the *Port Number* property in the Properties Window to 8080.

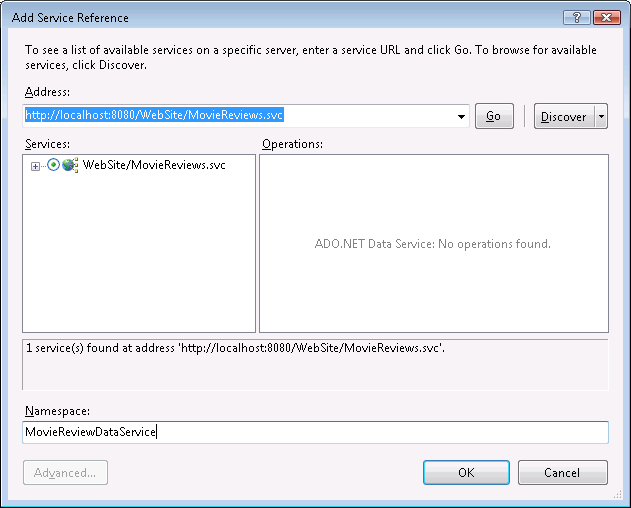
Going to the Properties Window twice works around an intermittent bug in Visual Studio. The Port Number will often stay in a read only mode until you exit and reenter.



1. Right-click the Solution in the Solution Explorer window and select *Properties*.
2. Under *Common Properties* -> *Startup Projects*, select the *Multiple Startup Projects* radio button. Under the *Action* header, set both projects in this solution to *Start*.
3. Right-click the DataServiceClient project in Solution Explorer and select “*Add Service Reference*”.
4. In the *Add Service Reference* dialog box, enter the address to the movie reviews service endpoint. The address should now be :

http://localhost:8080/WebSite/MovieReviews.svc

1. Enter a namespace of MovieReviewDataService, then click OK.



Visual Studio will now generate a proxy class we can use to query the data service. The name of the proxy will be the same as the server-side data source (in this case - MovieRepository).

1. Open *Program.cs* in the DataServiceClient project.
2. Add a *using* statement for *DataServiceClient*.*MovieReviewDataService* at the top of the file.
3. Our first task will be to query the data service. To do this we’ll need to instantiate a MovieRepository and give it the root URL to the data service. Add the following code inside the *Main* method.

Uri serviceRoot = new Uri(

"http://localhost:8080/WebSite/MovieReviews.svc");

MovieRepository repository = new MovieRepository(serviceRoot);

1. Our service is configured to use Windows authentication by default, so we will setup the proxy to send the current Windows credentials by adding the following line of code next:

repository.Credentials =

System.Net.CredentialCache.DefaultCredentials;

1. Finally, we are ready to issue a query to the service via the *MovieRepository* object we’ve instantiated. Use the *Movies* property of the repository to retrieve the 10 most recently released movies, then print the title of each movie.

var mostRecentMovies =

(from m in repository.Movies

orderby m.ReleaseDate descending

select m).Take(10);

foreach (var movie in mostRecentMovies)

{

Console.WriteLine(movie.Title);

}

1. We can also pull each movie’s reviews when we issue the query using the Expand operator. Use the expand operator in the query as shown below.

var mostRecentMovies =

(from m in repository.Movies**.Expand("Reviews")**

orderby m.ReleaseDate descending

select m).Take(10);

The data service proxy also supports an explicit “lazy-load” via the LoadProperty method.

1. Now, add some code to include the body of each review in the output for each movie.

foreach (var movie in mostRecentMovies)

{

Console.WriteLine(movie.Title);

**foreach (var review in movie.Reviews)**

**{**

**Console.WriteLine(review.Body);**

**}**

}

1. We can also update our reviews. Let’s add one to the rating of the first review in the first movie returned by this query. We’ll need to use SaveChanges on the repository to send the changes back to the server.

var firstReview = mostRecentMovies.First().Reviews.First();

firstReview.Rating = firstReview.Rating + 1;

repository.SaveChanges();

1. Press F5 to run both projects in the debugger and ensure you do not have any errors.

# Conclusion

Congratulations! You’ve implemented an ADO.NET Data Service complete with a service operation and a query interceptor. You’ve also written a .NET client that can consume the query in both read and update scenarios. If you want to move further ahead with ADO.NET data services, you can investigate how to delete and insert new reviews to the moviereviews database.