Introduction to Entity Framework

# I. Object Relational Mapping

Now that I've completed my [Introduction to LINQ](http://codetunnel.com/introduction-to-linq-part-i-delegates) tutorial I want to move on to another technology that takes Language Integrated Queries to a whole new level. [Entity Framework](http://msdn.microsoft.com/en-us/library/aa697427(v=vs.80).aspx) (EF) is what is known as an [Object Relational Mapper](http://en.wikipedia.org/wiki/Object-relational_mapping)(ORM). ORM is simply the concept of processing data from a relational database and modeling it in a format that is more readily usable by your code. Instead of looping through rows in a table you model each row as an object and loop through a collection of objects, called [entities](http://en.wikipedia.org/wiki/Entity).

## The Old Way

The simplest way to explain ORM is to first see how we would access data without it. Since this blog is .NET-centric I will show you how to access a database in C# the old fashioned way, using plain old [ADO.NET](http://en.wikipedia.org/wiki/ADO.NET). Here are the steps you would need in order to connect to a SQL server database:

1) First you need a SqlConnection object.

SqlConnection connection = new SqlConnection(connectionString);

If you've never seen a connection string before, it's just a string containing all the authentication data necessary to connect to the desired database. [Here is a sample](http://www.connectionstrings.com/sql-server-2008) of what a connection string looks like.

2) Once you have a connection, you need a SqlCommand object.

SqlCommand command = new SqlCommand(@"INSERT INTO [CodeTunnel].[Users]

VALUES (@Username, @Password, @Administrator)", connection);

3) Now we have to pass in the parameters by adding them to the Parameterscollection on the SqlCommand.

command.Parameters.Add(new SqlParameter("Username", "Alex"));

command.Parameters.Add(new SqlParameter("Password", "Exp3rtC0d3r1"));

command.Parameters.Add(new SqlParameter("Administrator", true);

4) Then you have to open the connection to the underlying database.

connection.Open();

5) Execute the command.

int rowsAdded = command.ExecuteNonQuery();

Console.WriteLine("{0} rows were added.", rowsAdded);

6) Finally you have to close the connection to the database.

connection.Close();

While ADO.NET greatly simplifies a lot of the steps necessary to connect to a database and execute queries, there are still several problems. First, all six steps here are required for every query you want to execute. In fact, an extra step is required if the query returns any data you want to collect; you have to create a SqlDataAdapter object and use that to populate a DataTable object. This is a giant pain to do with every single query. Adding a single row to a table requires quite a bit of work.

Another issue with old fashioned ADO.NET objects is that you have to include your SQL query in a string. As far as the .NET compiler is concerned your query is just a string. You won't know if you had any errors in your query until you run the application and execute the query. Only then will you get a SQL exception telling you something was wrong. This is not fun to debug, especially in large applications where you may not use that query very often and you won't know it's broken until months later when a visitor ends up executing it and getting an ugly error you didn't anticipate.

Another inconvenience with this method is the dependency on the developer knowing the syntax of another language. While I think learning SQL syntax is something every good developer should do, it is definitely an inconvenience for someone who may not be a SQL guru and just wants to build a small application that does some basic reading, inserting, and updating.

## The New Way

Today we have Object Relational Mappers. In .NET there are two different ORM products that do basically the same thing. The first one is [LINQ to SQL](http://msdn.microsoft.com/en-us/library/bb425822.aspx) (L2S). Second we have Entity Framework. L2S is the older of the two technologies. At their core they both accomplish the same goal, but L2S has quite a few more limitations than EF. The most obvious limitation is that L2S only works with a SQL Server database. Because EF is the more robust new kid on the block I'm going to focus this tutorial on that technology. Ultimately they are both just as easy to use and knowledge you gain while learning one can easily be translated to the other. Personally I feel that EF is the better technology and that L2S is slowly being phased out.

The ultimate goal of ORMs in .NET is to remove the need to execute string queries and give the developer the ability to write strongly-typed queries. If you're familiar with LINQ then you already know how to write strongly-typed queries against IEnumerable<T> collections. The benefit of writing a strongly-typed query is that the compiler can look at a LINQ query and tell you if you made a mistake. Of course now and then you'll still run into runtime errors, but the benefit of compiled queries is immense.

The ORM's job is to take your LINQ statements and translate them into SQL queries for you behind the scenes. This enables a developer to write complex queries against the database without ever having to write a single line of SQL. This does have the potential to allow SQL ignorant developers to remain ignorant but I think that is a small risk compared to the advantages it provides.

In the next parts of the tutorial I will go into detail about how to get going with Entity Framework but real quickly before I end Part I let's do a brief overview of the steps it would take to accomplish the same task that we did with plain ADO.NET. If you are unfamiliar with EF then this won't make tons of sense quite yet, but don't worry about it. Stay tuned for Parts II and III for more details.

1) Instantiate the data context object.

CodeTunnelEntities dataContext = new CodeTunnelEntities();

// EF will automatically read the connection string from your application configuration file :)

2) Create a new User object.

User newUser = new User

{

Username = "Alex",

Password = "Exp3rtC0d3r1",

Administrator = true

};

3) Add the user to the data context.

dataContext.Users.AddObject(newUser);

4) Save changes to the data context in order to persist those changes to the database.

dataContext.SaveChanges();

That's it! Obviously there is a lot more details you're missing in order to get started with EF and set up your model, but we'll cover that in the next part. For now you can sort of see the simplicity that the ORM provides. Remember how I said retrieving data in plain old ADO.NET would require an extra step? Well here retrieving data would actually be less steps! Here is an example of pulling out some data through a LINQ query.

List<User> administrators = dataContext.Users.Where(x => x.Administrator);

Could that be any easier? I think not.

# II. Designing the Database

## Database First vs Model First

There are two ways to design a model in Entity Framework (EF). Before we can build our model, you must first understand the difference between the two.

The first way to design your model in EF is to first design the database. If you have ever had any experience in setting up a SQL database before then nothing really changes in this step. Simply design your database as you normally would, keeping a few concepts in mind as you do it (we will go over these concepts shortly).

The second way to design your model in EF is to do the designing in EF itself and then instruct EF to build the database for you. This is an interesting route to take. It has the benefit of allowing the developer to again remain SQL ignorant (whether that is a good thing or not is debatable) as well as letting you use the EF design surface to map out your entities the way you like them. My issue with this approach is two-fold. Firstly, the paradigm of model first seems to me like starting from the middle. When I am designing my model in EF with no knowledge of the underlying database structure it becomes this weird disconnect; it is like framing a house and then instructing the frame to go ahead and pour the foundation for you.

The paradigm alone is not a huge knock against it as that would just be a matter of choice, however there is one technical issue I still have with the model first approach. When doing the model first EF will actually run a SQL script to setup your database. The issue with this is that if you create an entity in the designer and EF generates a corresponding table in your database, then your application stores some data in that table, and then you go to modify your entity, EF will re-generate your tables for that entity, wiping all associated data previously stored for that entity.

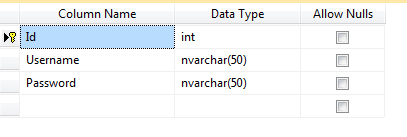
In my opinion, model first is still too buggy to be a very popular choice. There is one option that allows EF to update the database schema without wiping data, known as the [Entity Designer Database Generation Power Pack](http://blogs.msdn.com/b/adonet/archive/2010/02/08/entity-designer-database-generation-power-pack.aspx) (quite a mouthful). I toyed with this extension a little bit and while it is heading in the right direction, I just do not think it is quite there yet. It is also not very easy to figure out and setup. Personally, I would wait for Microsoft to make EF support these abilities natively before I would consider model first to be a viable approach.

For the purposes of this tutorial, I am going to instruct you how to do database first modeling with Entity Framework. It is currently the easiest approach to take and also the easiest to understand. If you have never dealt with a database before I am going to show you how to setup a very basic SQL Server Express database for your application using only Visual Studio. For larger applications, you would probably want to take advantage of a full-blown SQL Server instance and use the SQL Server Management Studio to design and make changes to your database, but for learning purposes SQL Server Express will do just fine.

## Designing the Database

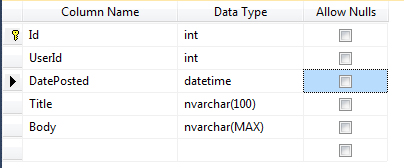
Before we can do anything with EF you will want to first design your underlying database. Follow the steps below to create an instance of a SQL Server Express Database and add some tables to it.

1. First open up Visual Studio and create a new console application. You can add Entity Framework models to any .NET application but for this tutorial, I am going to show you everything in a simple console application so as not to confuse you with out of scope knowledge about different application types.
2. Right-click on your project node in solution explorer and add a new item. Navigate until you find "Service-based Database". Let us name it "SampleDatabase.mdf and click "Add". Another screen will pop up asking you how you would like to model your database. We *could* go ahead and use this dialog to create our Entity Framework model file but for now just click "cancel", we will create that manually when we have done setting up our database. You should now have a SampleDatabase.mdf file in your solution explorer.
3. Double click your new database file and you should see the server explorer window appear and your sample database selected. Expand the node next to it and right-click "Tables". Select "Add New Table". You should now see a mostly blank screen with a small table containing three columns.
4. We are going to create a database for a fictitious blog application. The application will store users, blog posts, and post comments. Let us first create the table for our users. Modify your table to look like this:

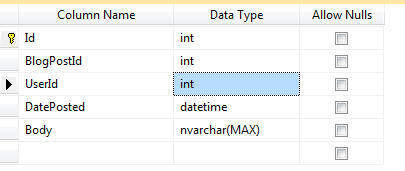


* + After putting in the data for the columns you need to set the "Id" column as the primary key. Right-click the space to the left of the column name and click "Set Primary Key". Next left click the space next to column name so as to select the whole column entry. Then down below in the "Column Properties" window navigate to the "Identity Specification" node and expand it. Set the "(Is Identity)" option to "Yes".
  + Save the table and name it "Users".

Follow the same procedure and add two more tables. Be sure to add the primary keys and set identities to yes.



* + Save the table and name it "BlogPosts".



* + Save the table and name it "PostComments".

1. The last thing we are going to do is add relationships between your tables. These relationships define how the data stored in the database is associated with one another. Open up your BlogPosts table, right-click an empty area of the design surface and go to "Relationships", then follow these instructions:
   * Click the "Add" button to generate a new relationship entry in the window.
   * In the properties on the right change name to be "FK\_UserHasManyBlogPosts".
   * Expand the "Tables And Columns Specification" node and click the button to the right of it to open up the relationship editor.
   * In the "Primary key table" dropdown select "Users".
   * Then in the row below that select the "Id" column.
   * Then on the right select "UserId" column.

You just designed a relationship that tells the database that a user could have many blog posts. Next open up your PostComments table and open the relationships window. Follow the same steps to create two more relationships.

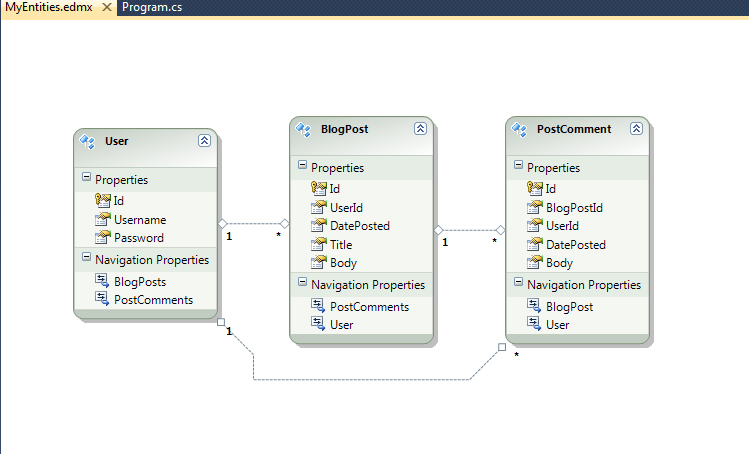
* + FK\_BlogPostHasManyPostComments
    - Primary key table: BlogPosts
    - Primary key column: Id
    - Foreign key column: BlogPostId
  + FK\_UserHasManyPostComments
    - Primary key table: Users
    - Primary key column: Id
    - Foreign key column: UserId

The hard part is over. Designing the underlying database can sometimes be tedious, but being intimately familiar with how your data is organized and stored is very valuable. While this has the potential to be a semi-confusing process you will see how much it greatly simplifies building the model in part three.

## Building the Model

Once you have your database setup and your relationships defined, setting up your model is a breeze. Let's go ahead and start building our model. Just follow these steps to create your Entity Framework model.

1. Right-click your project node and go to "Add" > "New Item".
2. From the list, choose "ADO.NET Entity Data Model". Name it "MyEntities.edmx" and click OK.
3. Next you'll see a screen asking you if you want to create an empty model or generate one from a database. Hopefully it's obvious to you that we want to create one from a database.
4. From the next screen you should see a drop down menu asking which data connection you should use to connect to the database. Expand the list and you should see the sample database we created in part two. Select it and check the box that says "Yes, include the sensitive data in the connection string" if it lets you. Then ensure that "Save entity connection settings in App.Config as" is checked. Leave the name the default. Click next.
5. On the next screen you should see three nodes. The first node should be "Tables" and should have an arrow to the left that you can expand. If you expand it you will see the tables we created. Check the box next to tables and click finish. You should see the EF design surface display with a visualization of your database as a model.



Notice that it even has a visual representation of the relationships you defined in the database relationships wizard.

## What Just Happened?

The model you just created is much more than just a simple diagram to represent the underlying database in a visual format. You generated an XML file that represents your database tables as entities, but what you may not realize is that you also generated a C# file called MyEntities.Designer.cs. This C# file is where your entities were created as C# objects. I will show you how to take advantage of this shortly, but it's important that you realize what you actually did just now. You actually generated C# code. All the work of creating C# objects that represent data from your database is gone. Entity Framework created those objects for you and now all you have to do is use them. This tool just saved you several hours of work.

## Now What?

Now we get to the fun part, actually using our entities in our application. To use your newly-generated entities you will need to become familiar with what is called an "entity container" or "data context". An entity container is simply an object created by the entity framework. This object contains collections of entities as properties. Using LINQ you are able to query these collections from within C# itself rather than writing SQL queries. To use your entity container you have to first instantiate it. Go ahead and open up the Program.cs file in your console application and modify the Main() method to look like this:

static void Main(string[] args)

{

// Instantiate your entity container.

MyEntitiesContainer container = new MyEntitiesContainer();

// Now lets create some sample users.

User sampleUser1 = new User

{

Username="Jakinator",

Password="IllBeBack"

};

User sampleUser2 = new User

{

Username="ILoveKittens",

Password="meow"

};

User sampleUser3 = new User

{

Username="SoccerMom",

Password="MyKidsAnHonorStudent"

};

// Now add the sample users to the entity container.

container.Users.AddObject(sampleUser1);

container.Users.AddObject(sampleUser2);

container.Users.AddObject(sampleUser3);

// We've added the users to the data context (entity container) but these changes won't

// be added to the database until we call the SaveChanges() method on the container.

container.SaveChanges();

}

In a relatively small amount of code we added three users to the database. Remember how hard that would have been if you did it with plain ADO.NET like we did in part one? The syntax here makes it really easy to see what's going on and severely decreases the amount of code you have to write. Before you run your program let's modify this a little more to add some blogs and post comments.

...

container.Users.AddObject(sampleUser1);

container.Users.AddObject(sampleUser2);

container.Users.AddObject(sampleUser3);

container.SaveChanges();

// Create a new blog post.

BlogPost samplePost = new BlogPost

{

DatePosted = DateTime.Now,

Title = "Why Terminator is the best movie ever!",

Body = @"Terminator is the best movie ever because it's the only one I've ever seen.

It was so awesome that when I finished watching it I vowed to never watch another movie again."

};

// Create sample comments for the above blog post.

PostComment sampleComment1 = new PostComment

{

UserId = sampleUser1.Id,

DatePosted = DateTime.Now,

Body = "First! I'm always the first to comment on my own blog posts. I'm just awesome like that."

};

PostComment sampleComment2 = new PostComment

{

UserId = sampleUser2.Id,

DatePosted = DateTime.Now,

Body = @"Jake I completely disagree that Terminator is the best movie ever for one reason:

There were no kittens in Terminator. If there were kittens, then maybe..."

};

PostComment sampleComment3 = new PostComment

{

UserId = sampleUser3.Id,

DatePosted = DateTime.Now,

Body = @"I would never let my kids watch horrible movies like this.

They should be at soccer practice or doing homework.

That's why my kids have all straight A's and are all honor students."

};

// Add the sample comments to the blog post's collection of post comments.

samplePost.PostComments.Add(sampleComment1);

samplePost.PostComments.Add(sampleComment2);

samplePost.PostComments.Add(sampleComment3);

// Add the blog post to the first sample user's collection of blog posts.

sampleUser1.BlogPosts.Add(samplePost);

// Persist changes to the entity container to the database.

container.SaveChanges();

This is a lot more code this time around. Hopefully you can see what's going on here. We created three users and added them to the container. We then created a blog post and three comments. Then we added the comments to the blog post and the blog post to the user. There are several things to take note of here. First, notice that we called container.SaveChanges() after we added our sample users. If you notice, we didn't set the Id property on our three user objects. The reason for this is that the Id column in the database has Is Identity set to yes; the database is supposed to create an Id for us. Until our changes are saved to the database our user objects would have no Id set. However, after calling container.SaveChanges() their Id properties get populated with the value the database used for them. This allows us to set the UserId on the post comments later in the code.

Now I bet you're wondering why we didn't set the UserId property on the blog post, or why we didn't set the BlogPostId property on the post comments. We used a neat little feature called navigation properties. If you look back up at the picture of our model you will see at the bottom of your entities that you have some navigation properties. Because EF was able to detect the relationships we set up, it knows that a user can have many blog posts, a blog post can have many comments, and a user can have many comments. It then creates these navigation properties for us. When you add a blog post to sampleUser.BlogPostsyou are automatically associating that blog post with that user. If you tried to add the blog post to the collection on the container container.BlogPosts.AddObject(newBlogPost) then you would have to set the UserIdproperty because it would not be able to populate it on its own. The same thing happens with post comments; by adding them to the blog post object the BlogPostId property is automatically set. We do have to set the UserId property though because we aren't adding it to a user object, though we very well could have done that and then set BlogPostId property ourselves.

Go ahead and run your program. Your console should pop up and then close again real fast. In that split second it took to run your application you added three users, a blog post, and three comments to the database. Pretty nifty eh? Let's see now how we get that information back out of the database. This is where LINQ comes in and saves the day.

## LINQ to Entities

Even though modeling our database and creating objects (entities) on the fly makes things a lot simpler, we've still only scratched the surface of the power of Entity Framework. So far, we have only used plain entity objects and their collections on the container to create and add some entities. We have not applied the power of LINQ to these collections. Before I get into a detailed explanation of how things work I want to show you a sample query.

Now that we have some data in our database, go ahead and remove all the code you added to the Main() method in your console application. Then modify the method to look like this:

static void Main(string[] args)

{

MyEntitiesContainer container = new MyEntitiesContainer();

User myUser = container.Users.Single(x => x.Username == "ILoveKittens");

foreach (PostComment postComment in myUser.PostComments)

{

Console.WriteLine();

Console.WriteLine("Comment by: {0}", myUser.Username);

Console.WriteLine("Date Posted: {0}", postComment.DatePosted);

Console.WriteLine("Comment: {0}", postComment.Body);

}

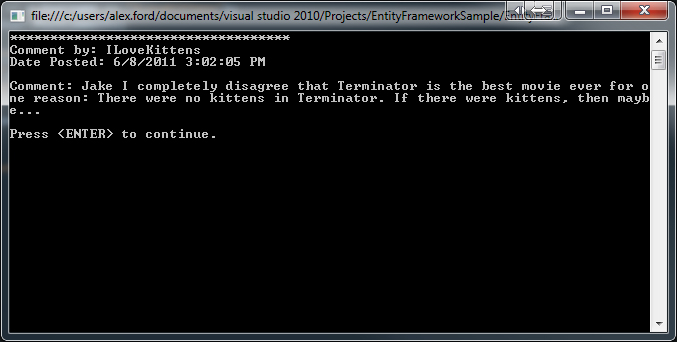
Console.WriteLine();

Console.WriteLine("Press <ENTER> to continue.");

Console.ReadLine();

}

It's super easy to read data in from Entity Framework. Here we created a Usercalled myUser and assigned it a user from the container via a LINQ query. We then looped through that user's post comments and if there were any then display them in the console window. If you followed the steps above to add data to the database then the result of running this should look like this:



Congratulations, you now know how to add data to the database as well as read it back. I will go into some more details about LINQ to Entities in part four of this tutorial, but if you need additional details in regards to LINQ in general then please see my [Introduction to LINQ](http://www.codetunnel.com/introduction-to-linq-part-i--delegates) tutorial. Before I end part three I want to show you quick samples of how to update and delete. First delete, because it's super easy.

User myUser = container.Users.Single(x => x.Username == "ILoveKittens");

container.Users.DeleteObject(myUser);

container.SaveChanges();

That's all you have to do to delete an entity. When I first started using EF it was the update part that really confused me. To update an entity all you have to do is change properties on the entity you retrieve from the container and then call container.SaveChanges(). At first I didn't understand why I didn't have to call something like container.Users.UpdateObject(myUser), but then I read about [self-tracking entities](http://blogs.msdn.com/b/efdesign/archive/2009/03/24/self-tracking-entities-in-the-entity-framework.aspx). Entities you retrieve from the container have code in them that tracks all the changes that are made to them. When you call SaveChanges()on the data context it already knows everything that changed and persists those changes to the database.

User myUser = container.Users.Single(x => x.Username == "ILoveKittens");

myUser.Username = "IStillLoveKittens";

container.SaveChanges();

That's all you have to do to update an entity.