# Recipe: Create Entity Framework Repository

## Overview

EF offer a simple object oriented solution to handle our data needs. You only need to conceptually define your data model and EF will create a relational database and do all the pluming for you. Then you only need to deal data object (entities) instead of SQL calls and parsing.

Creating an Entity Framework (EF) is very simple you just need:

1. A project that requires data management
2. Install the required software libraries
3. Create a model for your data
4. Create a context

## Preparations

You need an existing project. If you don’t have one here is an example:

* Open Visual Studio
* **File -> New -> Project…**
* Select **Windows** from the left menu and **Windows Form** **Application**
* Enter **a name** for your project.
* Select **OK**

Once Inside your project, use the Package Manager NuGet and install the lasted version of Entity Framework

## Create the Model of your data

EF is an extra layer that sits between your program and your database that handles all the complexities of the interactions with your database and gives you an object-oriented approach instead. These layers are commonly known as ORM (Object-Relational Mappers).

Instead of creating your database by creating your tables in SQL you will create classes that represent your data.

If you were to create a simple blog it would look like this:

public class Blog

{

public int BlogId { get; set; }

public string Name { get; set; }

public virtual List<Post> Posts { get; set; }

}

public class Post

{

public int PostId { get; set; }

public string Title { get; set; }

public string Content { get; set; }

public int BlogId { get; set; }

public virtual Blog Blog { get; set; }

}

## Create your context

For EF to work it requires a context. The context is a class you create that knows about your data. You will call this class whenever you need to interact with your data repository.

*Note: It is fundamental that you have installed the EF library in your project, otherwise this will not work.*

Create a class inside your project tat will act as your data context. It should look like this:

using System.Data.Entity;

public class BloggingContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

public DbSet<Post> Posts { get; set; }

}

Note that is Imperative that you reference the EF domain of the DbContext class. Otherwise .net does not know what DbContext is. You do this by writing the using statement followed by the domain of the class you are using at the beginning of your file.

What you are doing is create a class that inherits form an EF class that knows how to be a data context. Then all you need to do is expose your data collections.

# Recipe: Adding, Getting, Updating, and Deleting data to your Entity Framework Repository

## Overview

Manipulating data in and EF repository is extremely simple

## Prerequisites

In order to access your data you need to open a connection to your context and do all your operations inside it.

using (var db = new BloggingContext())

{

… stuff goes in here

}

## Adding data

Just create an object of the relevant class, fill it with information and add it to the repository.

For example, we know that we want to create blogs with post and that we have a blog class and a post class. Therefore, we create a new blog and a post and add them to the repository as shown in the code example bellow.

using (var db = new BloggingContext())

{

var blog = new Blog();

blog.Name = "Epic Blog";

db.Blogs.Add(blog);

var post = new Post();

post.Titel = "My first post";

post.Content = "I am so happy! ";

post.Blog = blog;

db.Posts.add(post);

db.SaveChanges();

}

## Query data

There are many ways to query your data from your repository using EF. The simplest ways are to leverage the power given by EF storing your data as object. You can use all the tools that .net gives you to retrieve objects.

As an example imagine you want to get all the posts from blog give

Your context exposes two collections of data: Blogs and Posts. You can get data out of a collection using the functional approach:

using (var db = new BloggingContext())

{

var blog = db.Blogs.First(b => b.Name == " Epic Blog");

var posts = blog.Posts;

}

Or using LINQ to use a more familiar SQL syntax:

using (var db = new BloggingContext())

{

var q = from b in db.Blogs

where b.Name == "Epic Blog"

select b.Posts;

}

## Update Data

Updating data is as simple as query the data you want to update, update it and save it back

For example if we want to change the name of our “Epic Blog” to “Nice Blog” we would do:

using (var db = new BloggingContext())

{

var blog = db.Blogs.First(b => b.Name == "Epic Blog");

blog.name = "Nice Blog";

db.SaveChanges();

}

## Delete data

Deleting data works the same as update.

For example if we want to delete our “Nice Blog” we would do:

using (var db = new BloggingContext())

{

var blog = db.Blogs.First(b => b.Name == "Nice Blog");

db.Blogs.Remove(blog)

db.SaveChanges();

}

# Recipe: Reading data from text files

## Overview

There a many ways to achieve this goal, but for the purposes of a plug-and-play approach we will use a third party library called LINQtoCSV that abstracts the task into an object oriented approach.

To parse (analysis a string of symbols) your file with LINQtoCSV you need:

1. Create a class that describes your text files
2. Create a File description object
3. Indicate the location and name of your file
4. Initialize your reader
5. Iterate your file

## Preparations

We will need out 3rd party library LINQtoCSV to use the Package Manager NuGet and install the lasted version of LINQtoCSV.

## Reading a text file

As mentioned earlier LINQtoCSV uses and object oriented approach to reading text files. First, we need to create a class that **precisely** describes your text file.

Assume we have the following text file:

Blog Name,PostTitel,PostContent

Epic,Fisrt post, ABC

Epic,Second post, DEF

Lost,Fisrt, ZDF

We will need a class like:

using LINQtoCSV;

public class PostImport

{

[CsvColumn (Name = "Blog Name")]

public string BlogName { get; set; }

[CsvColumn]

public string PostTitel { get; set; }

[CsvColumn]

public string PostContent { get; set; }

}

Notice the bracketed attribute “[CsvColumn]”. This attribute (or decoration as they are technically known) will tell LINQtoCVS how to match the text file columns to the objects created by this class. This means that you need to make sure that your text file column names match the names of the properties in your class. This decoration also take parameters like Name or FieldIndex. This are very useful to help you face different scenarios.

It is worth mentioning some of them:

*[CsvColumn (Name = "Blog Name")]*: the Name parameter will match the name of the text file column to the property. If no name parameter is specified, it will match the column with the property but this will fail if they are not identical. It is important to notice that property names cannot have spaces, so a column named “blog name” can not be map to a property unless you use this parameter.

*[CsvColumn(FieldIndex = 1)]*: If the text file has no headers to use for the mapping, the procedure is to set the what property maps to what column by using the FieldIndex parameter.

Additionally you will need to:

* Put in a variable the name and location of your text file
  + var file = “c: \Textfile.txt”;
* Create a CsvFileDescription object
  + var inputFileDescription = new CsvFileDescription();

inputFileDescription.EnforceCsvColumnAttribute = true;

inputFileDescription.FirstLineHasColumnNames = true;

inputFileDescription.SeparatorChar = ',';

IgnoreUnknownColumns = true;

* Initiate your reader
  + var cc = new CsvContext();

var data = cc.Read<PostImport>(file, inputFileDescription);

This will look like this:

var file = “c:\textfile.txt;

var inputFileDescription = new CsvFileDescription();

inputFileDescription.EnforceCsvColumnAttribute = true;

inputFileDescription.FirstLineHasColumnNames = true;

inputFileDescription.SeparatorChar = ',';

inputFileDescription.IgnoreUnknownColumns = true

var cc = new CsvContext();

var data = cc.Read<PostImport>(file, inputFileDescription);

You are done! All you need it’s to iterate thru your data:

foreach (var item in data)

{

var aPost = new Post();

aPost.Title = item.PostTitel;

aPost.Content = item.PostContent;

...

}

# Recipe: Advanced scenarios

## Overview

Here we will cover some advanced scenarios for the other protocols.

## Read from multiple files

The domain System.IO has many handy tools that allows us to play with files. One of such tools is the Directory.EnumerateFiles() function. You can use it this way:

using System.IO;

…

var dir = "c:\\Neo4j\\";

var fileList = Directory.EnumerateFiles(dir, "\*.tab");

foreach (var file in fileList)

{

}

## Connecting a query to a grid

Windows forms offers you a many ways to display your data one of them is the DataGridView. With this tools all you need to do is drag and drop a DataGridView into your form and give it a datasource to display

using (var db = new EntityContext())

{

//query using lambda expressions  
 var postsFromBlogLambda = db.Blogs.First(b => b.Name == "Big Blog").Posts.Select(n=>new {Title = n.Title, Content = n.Content});

//query using LINQ  
 var postFromBlogLinq = from b in db.Blogs  
 join p in db.Posts on b equals p.Blog  
 where b.Name == "Big Blog"  
 select new {p.Title, p.Content};

dgvTest.DataSource = postFromBlogLinq.ToList();  
 //or  
 dgvTest.DataSource = postsFromBlogLambda.ToList();

}

## Exporting data to text files

There are many ways to export data from your system. One of them is to export it as text files. This is a very simple procedure:

1. query your data
2. create a string variable and concatenate your values
3. Save your string data into a file

At the end, it will look like this:

var outputText = new StringBuilder();

using (var db = new EntityContext())  
{

//query using lambda expressions  
 var postsFromBlogLambda =  
 db.Blogs.First(b => b.Name == "Big Blog")  
 .Posts.Select(n => new {Title = n.Title, Content = n.Content})  
 .ToList();

//create headers  
 outputText.Append("Title").Append("\t");

outputText.Append("Content").Append("\t");

//add rows  
 foreach (var item in postsFromBlogLambda)  
 {  
 outputText.Append(item.Title).Append("\t");  
 outputText.Append(item.Content).Append("\t");  
 outputText.AppendLine();

}

var outputDirectory = "c:\\files";  
 using (var sw = new StreamWriter(outputDirectory))  
 sw.Write(outputText);

}