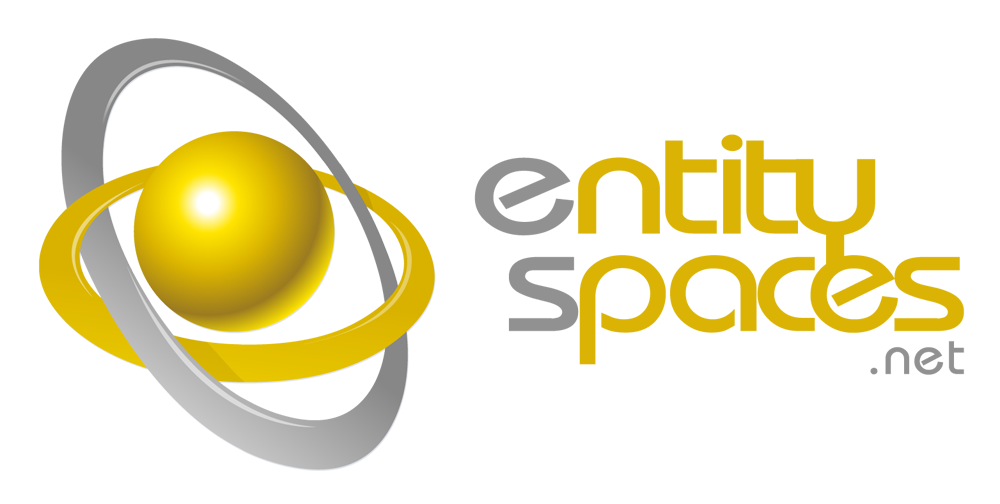
Collections

Using EntitySpaces Collections

November 19, 2011  
EntitySpaces, LLC  
The EntitySpaces Team

**[](http://www.entityspaces.net/)**

Persistence Layer and Business Objects for Microsoft .NET

Contents

[Collections 1](#_Toc317275961)

[LoadAll 1](#_Toc317275962)

[Loading a Collection using a Dynamic Query 1](#_Toc317275963)

[Iterating over a Collection 1](#_Toc317275964)

[The foreach Syntax 2](#_Toc317275965)

[The Index Syntax (slow way to iterate) 2](#_Toc317275966)

[RowState 2](#_Toc317275967)

[Adding a new Entity to a Collection 3](#_Toc317275968)

[Deleting Entities from a Collection 3](#_Toc317275969)

[MarkAsDeleted 3](#_Toc317275970)

[MarkAllAsDeleted (the wrong way) 3](#_Toc317275971)

[MarkAllAsDeleted (the right way) 4](#_Toc317275972)

[The DeletedEntities List 4](#_Toc317275973)

[Saving Data 4](#_Toc317275974)

[A Simple Save (commit or fail as a unit) 5](#_Toc317275975)

[A Partial Save (continueUpateOnError) 5](#_Toc317275976)

[RejectChanges and AcceptChanges 6](#_Toc317275977)

[RejectChanges 6](#_Toc317275978)

[AcceptChanges 6](#_Toc317275979)

[Filtering and Sorting 6](#_Toc317275980)

[A Simple Filter 7](#_Toc317275981)

[A Simple Sort 7](#_Toc317275982)

[Filter and Sort Together 7](#_Toc317275983)

[DetachEntity and AttachEntity 8](#_Toc317275984)

[Combine 8](#_Toc317275985)

[Using LINQ to Query against a Collection 9](#_Toc317275986)

[Advanced Loading Techniques 10](#_Toc317275987)

[Raw Text 10](#_Toc317275988)

[Raw Text with Parameters 11](#_Toc317275989)

[Stored Procedure 11](#_Toc317275990)

[Other Protected Methods to Access Data 12](#_Toc317275991)

[ExecuteNonQuery (with an OUTPUT parameter) 12](#_Toc317275992)

[ExecuteScalar 14](#_Toc317275993)

[ExecuteReader 14](#_Toc317275994)

# Collections

EntitySpaces collections are simply a collection of single EntitySpaces entities. A collection can be used to load one or more entities from the database and save the data back again. Collections support data binding, filtering, sorting, adding new entities, deleting entities, and many more operations which will be covered in this section.

## LoadAll

LoadAll does exactly what you think it would. It loads all of the records for the given collection type, in this case, all of the employees. Any method which loads data in EntitySpaces returns a Boolean indicating whether or not a least one record was returned. This way any load method can be wrapped with an “if” statement to determine whether or not any data was loaded.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

// Then we loaded at least one record

}

## Loading a Collection using a Dynamic Query

The full Dynamic Query API is covered in its own section. This is just a quick example to make you aware of the feature.

EmployeesQuery query = new EmployeesQuery();

query.Select(query.EmployeeID, query.FirstName, query.LastName);

query.Where(query.LastName.Like("%a%"));

EmployeesCollection coll = new EmployeesCollection();

if (coll.Load(query))

{

// Then we loaded at least one employee who

// has a LastName with an "a" in it

}

## Iterating over a Collection

There are two ways to iterate over collections. The first being the foreach syntax and the second by index. The foreach syntax is by far the preferred way and very fast. By index is a slow way to iterate over a collection. Accessing entities by index has its uses, but when you need to iterate over the collection use the foreach syntax.

### The foreach Syntax

The foreach syntax is the preferred way to iterate over a collection.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

// The preferred way to iterate, FAST

foreach (Employees employee in coll)

{

}

}

### The Index Syntax (slow way to iterate)

The index syntax is a **very slow** way to iterate over a collection. Accessing data by index is useful, but don’t use indexing to iterate over a collection. The example shown below is for education purposes.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

// Very SLOW way to iterate

for (int i = 0; i < coll.Count; i++)

{

Employees employee = coll[i];

}

}

## RowState

It is important to understand the concept of RowState. Every entity in a collection has a RowState value. The RowState is an enum and is as follows:

public enum esDataRowState

{

Invalid = 0,

Unchanged = 2, // This indicates that an entity has not been modified

Added = 4, // This indicates a new entity (INSERT)

Deleted = 8, // This indicates an entity has been marked as deleted (DELETE)

Modified = 16, // This indicates an entity has been modified (UPDATE)

}

An Entity should never have a RowState of Invalid. The RowState value is accessed via the RowState property on a single entity.

## Adding a new Entity to a Collection

Adding a new entity to a collection is achieved by calling the AddNew method. When you add an entity to a collection, and later call Save, the newly added entity will result in a new record being inserted (a SQL INSERT statement) into the database.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

Employees employee = coll.AddNew();

// Fill out the employee properties here ...

}

## Deleting Entities from a Collection

Entities are “marked” as deleted in a collection and kept in a holding pen within the collection. Once an entity is marked as deleted, it will no longer be visible when iterating over a collection. Entities marked as deleted are only truly deleted from the database if Save is called on the collection. Any entity marked as deleted will result in a SQL DELETE statement being executed for that entity during a call to Save.

There are two ways entities can be marked as deleted in a collection.

### MarkAsDeleted

This method marks a single entity for deletion.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

Employees employee = coll[0];

employee.MarkAsDeleted();

}

### MarkAllAsDeleted (the wrong way)

You should never modify the contents of a collection when iterating over it. This is not only true for EntitySpaces collections, but for all Microsoft .NET collections. Calling MarkAsDeleted on an entity in a collection removes the entity from the list of entities, that is why you cannot use MarkAsDeleted in a foreach loop.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

foreach (Employees employee in coll)

{

// Unpredictable results because we are in a foreach !!!

employee.MarkAsDeleted();

}

}

### MarkAllAsDeleted (the right way)

This is the correct way to mark all entities in a collection as deleted.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

coll.MarkAllAsDeleted();

}

### The DeletedEntities List

It is possible to iterate over the deleted entities list although you should really never need to do so. We are just listing this here for completeness. Never take any action on an entity in the DeletedEntities list.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

coll.MarkAllAsDeleted();

foreach (Employees employee in coll.es.DeletedEntities)

{

}

}

## Saving Data

When you want to save all of the modified records in a collection, all that is required is call to Save. The collection will internally declare a transaction and make sure the transaction commits or is rolled back as a single unit. The collection will save all new, deleted, and modified records.

Any primary key values that are identity/auto-increment/sequence-based will have those populated. Their values will be in the entity's primary key field after save.

### A Simple Save (commit or fail as a unit)

This sample saves a new entity, a modified entity, and deletes an entity.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

// Mark the first employee for deletion

coll[0].MarkAsDeleted();

// Add a new employee

Employees employee = coll.AddNew();

employee.FirstName = "Joe";

employee.LastName = "Smith";

// Modify an existing employee

employee = coll[4]; // Grab a random employee

employee.FirstName = "Sam";

employee.LastName = "Jones";

// Save all new, modified, and deleted employees

// in a transaction (no need to declare a transaction)

coll.Save();

}

### A Partial Save (continueUpateOnError)

Notice in the sample below "true" is passed to Save indicating to “continueUpdateOnError”. Passing in "true" ensures that errors do not stop the save process. Any records that are valid are saved even though other records may fail. The collection Errors property can then be checked to see if there were any errors. In the sample below, the new record is saved, the modified record fails to save.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

// Add a new employee (this will save)

Employees employee = coll.AddNew();

employee.FirstName = "Joeee";

employee.LastName = "Smithhh";

// Modify an existing employee

employee = coll[1];

employee.FirstName = null; // this field is required, will fail

employee.Title = "CHANGE TO MAKE SURE THIS IS THE ONE";

// Saves all new, modified, and deleted employees

// in a transaction (no need to declare a transaction)

coll.Save(true);

// Iterate over any errors

foreach (Employees emp in coll.Errors)

{

string title = emp.Title;

string error = emp.es.RowError;

}

}

The value of emp.es.RowError above is as follows for Northwind SQL Server:

*Cannot insert the value NULL into column 'FirstName', table 'Northwind.dbo.Employees'; column does not allow nulls. UPDATE fails. The statement has been terminated.*

## RejectChanges and AcceptChanges

Of these two methods RejectChanges is probably the most useful. It can be used to undo or cancel any changes made to a collection. AcceptChanges on the other hand is typically not called by the developer; rather EntitySpaces calls AcceptChanges after a successful call to Save.

### RejectChanges

RejectChanges can be very useful for implementing an Undo feature or a Cancel Changes feature. Reject changes reverts the collection back to its original state as it was when it was first loaded, or the state since the last call to RejectChanges.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

// Mark the first employee for deletion

coll[0].MarkAsDeleted();

// Add a new employee

Employees employee = coll.AddNew();

// Modify an existing employee

employee = coll[4]; // Grab a random employee

employee.FirstName = "Sam";

// Undo the AddNew, MarkAsDeleted, and modification

coll.RejectChanges();

}

At this point, after the call to RejectChanges, the collection is back to its original state. The new entity has been removed, the deleted entity has now been moved from the DeletedEntities list, and the changes made to the FirstName have been undone. The RowState of each entity is now “Unchanged”.

### AcceptChanges

There is no sample for this. EntitySpaces calls AcceptChanges after a successful call to Save. However, on the single Entity class AcceptChanges does have some applications.

## Filtering and Sorting

Filtering and sorting are performed on collections after the collection has been populated. To filter or sort the LINQ Lambda syntax expression is used.

### A Simple Filter

This filter simply filters the collection so that it contains only those entities which have an ‘a’ in their FirstName.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

// Filter on FirstName containing an "a"

coll.Filter = coll.AsQueryable().Where(d => d.FirstName.Contains("a"));

foreach (Employees employee in coll)

{

// Each employee's FirstName has an 'a' in

}

// Clear the filter

coll.Filter = null;

foreach (Employees employee in coll)

{

// All employees are now back in the list

}

}

### A Simple Sort

Sorting is done through the Filter property (see the example above). Below is a very simple sorting example. Sorting effects how data is ordered. This effects the foreach() syntax and databinding. When you iterate over a collection using foreach the entities are iterated over in the order defined by the sort.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

// Order by last name ascending

coll.Filter = coll.AsQueryable().OrderBy(d => d.LastName);

// Order by last name descending

coll.Filter = coll.AsQueryable().OrderByDescending(d => d.LastName);

}

### Filter and Sort Together

You can both filter and sort in a single statement as follows.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

// Filter where FirstName contains "a" and order ascending by LastName

coll.Filter = coll.AsQueryable().Where(d => d.FirstName.Contains("a"))

.OrderBy(d => d.LastName);

}

## DetachEntity and AttachEntity

These methods are complimentary. An entity can be detached from one collection and attached to another collection. However, do not manually create entities and call AttachEntity to add entities to a collection. Not only is it slower to do so but it will not work in with the hierarchical model.

The following sample detaches two entities from a collection and attaches them to another collection. The “otherCollection” has a count of two after the code snippet is complete.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

EmployeesCollection otherCollection = new EmployeesCollection();

Employees e1 = coll[0];

Employees e2 = coll[1];

otherCollection.AttachEntity(coll.DetachEntity(e1));

otherCollection.AttachEntity(coll.DetachEntity(e2));

}

## Combine

This method transfers the contents from a source collection to a destination collection. It is important to note that those entities transferred into the destination collection are removed from the source collection.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

EmployeesCollection otherCollection = new EmployeesCollection();

// This will transfer all of the entities into another collection

otherCollection.Combine(coll);

coll.Filter = null;

}

## Using LINQ to Query against a Collection

EntitySpaces collections allow you to execute LINQ queries against them. Below is a simple example which iterates over Employee entities.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

// All employees with "i" in the last name ordered by FirstName in decending order

var emps = from e in coll

where e.LastName.Contains("i")

orderby e.FirstName descending

select e;

foreach (Employees emp in emps)

{

Console.WriteLine(emp.FirstName + ", " + emp.LastName);

}

}

This example derives its own result set by using the “select new” syntax.

EmployeesCollection coll = new EmployeesCollection();

if (coll.LoadAll())

{

// All employees with "i" in the last name ordered by FirstName in decending order

var emps = from e in coll

where e.LastName.Contains("i")

orderby e.FirstName descending

select new { e.FirstName, e.LastName };

foreach (var dude in emps)

{

Console.WriteLine(dude.FirstName + ", " + dude.LastName);

}

}

This is all LINQ at work here, the EntitySpaces collections support the proper interface to allow LINQ to operate on them

## Advanced Loading Techniques

This section contains some examples of advanced loading techniques.

### Raw Text

This example uses a raw SQL string to populate the collection. Notice we use esQueryType.Text. One of the best features of EntitySpaces is its DynamicQuery API which doesn’t rely on hard coded SQL. Only use the feature below when the DynamicQuery is unable to provide what you need.

public partial class EmployeesCollection : esEmployeesCollection

{

public EmployeesCollection() { }

public bool CustomLoadWithRawText()

{

return this.Load(esQueryType.Text, "SELECT \* FROM [Employees] WHERE TITLE LIKE '%A%");

}

}

In our Custom class above we made our CustomLoadWithRawText method return a Boolean to follow the EntitySpaces pattern. All that needs to be done now is instantiate a collection and call our custom method.

EmployeesCollection coll = new EmployeesCollection();

if (coll.CustomLoadWithRawText())

{

// Then we found at least one record

}

### Raw Text with Parameters

This approach combines raw text with parameters. Place holders such as {0}, {1}, are used in the raw text to indicate where parameters are to be created. EntitySpaces will automatically create parameters for each placeholder for you thus thwarting any SQL Injection techniques. Notice the third parameter to this overload of Load() is an **params object[]**.

public partial class EmployeesCollection : esEmployeesCollection

{

public EmployeesCollection() { }

public bool CustomLoadRawTextWithParams(int employeeIdMinimum, string lastNameLike)

{

return this.Load(esQueryType.Text,

"SELECT \* FROM [Employees] WHERE [EmployeeId] > **{0}** AND LastName LIKE **{1}**",

employeeIdMinimum, lastNameLike);

}

}

Now all that needs to be done is invoke our custom method as follows.

EmployeesCollection coll = new EmployeesCollection();

if (coll.CustomLoadRawTextWithParams(42, "%a%"))

{

// Then we found at least one record

}

### Stored Procedure

Below we have a simple stored procedure that takes two parameters. The following illustrates how it can be used to load a collection.

CREATE PROCEDURE [dbo].[**Employee\_CustomLoad**]

(

@EmployeeIdMinimum INT,

@LastNameLike NVARCHAR(20)

)

AS

BEGIN

SET NOCOUNT ON

DECLARE @Err int

SELECT \*

FROM [dbo].[Employees]

WHERE [EmployeeID] > @EmployeeIdMinimum AND [LastName] LIKE @LastNameLike

SET @Err = @@Error

RETURN @Err

END

The custom load method takes two values which will be passed straight through to the stored procedure above.

public partial class EmployeesCollection : esEmployeesCollection

{

public EmployeesCollection() { }

public bool CustomLoadWithStoredProcedure(int employeeIdMinimum, string lastNameLike)

{

// Notice do not use @ or other symbols on your parameter names.

// The EntitySpaces providers will do this for you, this way even your

// stored procedure calls are database independent.

esParameters parms = new esParameters();

parms.Add("EmployeeIdMinimum", employeeIdMinimum);

parms.Add("LastNameLike", lastNameLike);

return this.Load(esQueryType.StoredProcedure, "**Employee\_CustomLoad**", parms);

}

}

And finally our invocation of the custom method.

EmployeesCollection coll = new EmployeesCollection();

if (coll.CustomLoadWithStoredProcedure(42, "%a%"))

{

// Then we found at least one record

}

## Other Protected Methods to Access Data

The examples in this section cover the special protected methods that may be used in your custom classes.

### ExecuteNonQuery (with an OUTPUT parameter)

ExecuteNonQuery is used to execute a SQL Statement or Stored Procedure without returning a result set. The stored procedure below takes three parameters; the first two are input parameters and the third is an output parameter. The first two parameters admittedly make little sense, but the next section will explain why we included them.

CREATE PROCEDURE [dbo].[Employee\_GetMostRecentHireDate]

(

@EmployeeIdMinimum INT,

@LastNameLike NVARCHAR(20),

@HireDate DATETIME **OUTPUT** -- < output parameter

)

AS

BEGIN

SET NOCOUNT ON

DECLARE @Err int

SELECT @HireDate = MAX(HireDate)

FROM employees

WHERE [EmployeeID] > @EmployeeIdMinimum AND [LastName] LIKE @LastNameLike

AND [HireDate] IS NOT NULL

SET @Err = @@Error

RETURN @Err

END

Below is the method that was added to the Custom class which invokes the stored procedure above. The reason we included the first two parameters was to show that you only need to use the more verbose version of esParameters.Add when have a special need, such as indicating a parameter is an output parameter. Most of the time all you need to do is pass the parameter name and value, as is done for the first two parameters. Finally, notice how we access the value of the output parameter by access the value of the HireDate parameter after the call to ExecuteNonQuery.

public partial class EmployeesCollection : esEmployeesCollection

{

public EmployeesCollection() { }

public DateTime? GetMostRecentHireDate(int employeeIdMinimum, string lastNameLike)

{

DateTime? hireDate = null;

// Notice do not use @ or other symbols on your parameter names.

// The EntitySpaces providers will do this for you, this way even your

// stored procedure calls are database independent.

esParameters parms = new esParameters();

parms.Add("EmployeeIdMinimum", employeeIdMinimum);

parms.Add("LastNameLike", lastNameLike);

parms.Add("HireDate", esParameterDirection.Output, System.Data.DbType.DateTime, 0);

// ExecuteNonQuery only execute SQL it doesn't load a collection with data

this.ExecuteNonQuery(esQueryType.StoredProcedure,   
 "Employee\_GetMostRecentHireDate", parms);

// **Get the output parameter value**

hireDate = (DateTime?)parms["HireDate"].**Value**;

return hireDate;

}

}

Here is the invocation of the custom method which returns the most recent hire date.

EmployeesCollection coll = new EmployeesCollection();

DateTime? maxHireDate = coll.GetMostRecentHireDate(5, "%a%");

### ExecuteScalar

ExecuteScalar returns a single value. The sample below returns the count of the employees with a particular last name.

public partial class EmployeesCollection : esEmployeesCollection

{

public EmployeesCollection() { }

public int GetCountOfEmployeesWithThisName(string lastName)

{

string sqlText =

"SELECT COUNT([FirstName]) FROM [Employees] WHERE [LastName] = {0}";

return (int)this.ExecuteScalar(esQueryType.Text, sqlText, lastName);

}

}

Here is the invocation of the custom method which returns the count.

EmployeesCollection coll = new EmployeesCollection();

int count = coll.GetCountOfEmployeesWithThisName("Griffin");

### ExecuteReader

This method will return a DataReader that you are responsible for closing. This method in our custom class returns a DataReader for all employees.

public partial class EmployeesCollection : esEmployeesCollection

{

public EmployeesCollection() { }

public IDataReader GetReaderForAllEmployees()

{

return this.ExecuteReader(esQueryType.Text, "SELECT \* FROM [Employees]");

}

}

This code will invoke the custom method above making sure to use the “using” syntax to close the DataReader. The method below does a Console.WriteLine printing each employee's last name to the console.

EmployeesCollection coll = new EmployeesCollection();

using (IDataReader rdr = coll.GetReaderForAllEmployees())

{

int ordinal = rdr.GetOrdinal("LastName");

while (rdr.Read())

{

Console.WriteLine(rdr.GetString(ordinal));

}

}