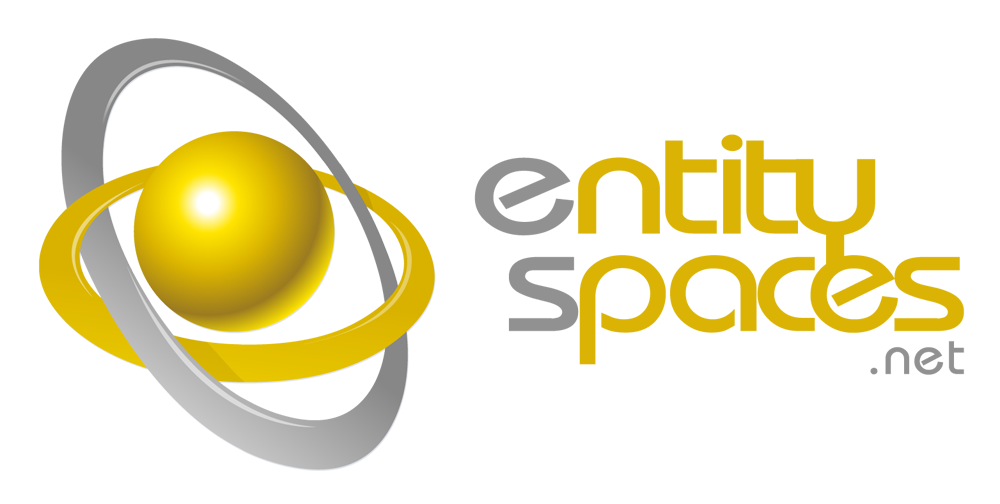
Entity

Using EntitySpaces entities

February 11, 2012  
EntitySpaces, LLC  
The EntitySpaces Team

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

Persistence Layer and Business Objects for Microsoft .NET

Contents

[Entities 1](#_Toc317319026)

[LoadByPrimaryKey 1](#_Toc317319027)

[RowState 1](#_Toc317319028)

[Adding a new Entity and Saving It 2](#_Toc317319029)

[Loading an Entity and Updating It 2](#_Toc317319030)

[Loading an Entity using a Dynamic Query 2](#_Toc317319031)

[Deleting an Entity 3](#_Toc317319032)

[MarkAsDeleted 3](#_Toc317319033)

[Deleted (static method) 3](#_Toc317319034)

[RejectChanges and AcceptChanges 3](#_Toc317319035)

[RejectChanges 3](#_Toc317319036)

[AcceptChanges 4](#_Toc317319037)

[Various Helper Routines 4](#_Toc317319038)

[ContainsColumn and GetOriginalColumnValue 4](#_Toc317319039)

[Advanced Loading Techniques 5](#_Toc317319040)

[Raw Text 5](#_Toc317319041)

[Raw Text with Parameters 5](#_Toc317319042)

[Stored Procedure 6](#_Toc317319043)

[Other Protected Methods to Access Data 7](#_Toc317319044)

[ExecuteNonQuery (with an OUTPUT parameter) 7](#_Toc317319045)

[ExecuteScalar 8](#_Toc317319046)

[ExecuteReader 9](#_Toc317319047)

# Entities

Entities represent a single row in the database. They are also contained in collections. You can load and save an entity without using a collection. Entities can also be loaded using the Dynamic Query API. If you try to load more than one record into an entity an exception is thrown.

EntitySpaces Studio does not tack on an “s” to pluralize your entites. The examples used in the following document are from the Microsoft SQL Northwind database, specifically the ‘Employees’ table. Thus, the “s” on the end of Employees exists only because that is the actual table name.

## LoadByPrimaryKey

LoadByPrimaryKey does exactly what you think it would. It loads a record using the primary key. 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.

Employees employee = new Employees();

if (employee.LoadByPrimaryKey(2))

{

// Then it was loaded

}

## RowState

It is important to understand the concept of RowState. Every entity 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 since being loaded

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. There are also helper functions available to check RowState as well.

Employees employee = new Employees();

bool isAdded = employee.es.**IsAdded**;

bool isDeletd = employee.es.**IsDeleted**;

bool isModified = employee.es.**IsModified**;

bool isDirty = employee.es.**IsDirty**; // True if any of the above are true

## Adding a new Entity and Saving It

Adding a new entity and then saving it to the database is very simple. This will result in a new record being inserted (using a SQL INSERT statement) into the database.

Employees employee = new Employees();

employee.FirstName = "Joe";

employee.LastName = "Smith";

employee.HireDate = DateTime.Now;

employee.Save();

## Loading an Entity and Updating It

Updating an existing record is easy, simply load it, change the desired properties, and save it.

Employees employee = new Employees();

if (employee.LoadByPrimaryKey(2))

{

employee.FirstName = "Joe";

employee.LastName = "Smith";

employee.Save();

}

Every EntitySpaces has a default setting that indicates whether you are using “Stored Procedures” or “Dynamic SQL”. It is possible to override this during Save. For instance, if your default setting is to use “Dynamic SQL” and let EntitySpaces build your UPDATE statement. To override the Save() method you would do so like this:

Employees employee = new Employees();

if (employee.LoadByPrimaryKey(2))

{

employee.FirstName = "Joe";

employee.LastName = "Smith";

employee.Save(**esSqlAccessType.StoredProcedure**);

}

Notice we pass in a value to Save indicating that we want to use a stored procedure to Save in this one case. EntitySpaces has templates that can generate your CRUD (create, read, update, delete) stored procedures.

## Loading an Entity 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. If you plan on updating it, remember you need to bring back the primary key(s) in the query.

EmployeesQuery query = new EmployeesQuery();

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

query.Where(query.EmployeeID == 42);

Employees employee = new Employees();

if (employee.Load(query))

{

// Then it was loaded

}

Remember, entities can only contain a single row so when writing a query that will be used to load a single entity keep that in mind.

## Deleting an Entity

Entities are “marked” as deleted and then must be saved to actually cause the delete to take place. An 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 to delete an entity.

### MarkAsDeleted

This method deletes a single entity.

Employees employee = new Employees();

if (employee.LoadByPrimaryKey(2))

{

employee.MarkAsDeleted();

employee.Save();

}

### Deleted (static method)

If you want to delete a record without first loading it, you can call the static Delete method. The one caveat with this approach is it will not work with tables that have a concurrency column in them.

Employees.Delete(42);

## 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 an entity. 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 entity back to its original state as it was when it was first loaded, or the state since the last call to RejectChanges.

Employees employee = new Employees();

if (employee.LoadByPrimaryKey(2))

{

employee.FirstName = "Joe";

employee.LastName = "Smith";

employee.RejectChanges(); // FirstName/LastName revert

}

At this point, after the call to RejectChanges the entities properties are back to their original state. The RowState of the entity is now “Unchanged” instead of “Modified”.

### AcceptChanges

EntitySpaces calls AcceptChanges after a successful call to Save. There are a few tricks you can do with AcceptChanges which are generally frowned upon and not considered best practices. For instance, you could update an existing record without first loading it from the database like this:

Employees employee = new Employees(); // Rowstate is ‘Added’ at this point

employee.EmployeeID = 42;

employee.AcceptChanges(); // RowState is ‘Unchanged’ at this point

employee.FirstName = "Joe"; // RowState switches to ‘Modified’

employee.LastName = "Smith";

employee.Save(); // Results in a SQL UPDATE

Again, we do not recommend the syntax above and you will run into trouble with tables that have concurrency columns using this approach. It is shown here to give you a clear understanding of what AcceptChanges does internally.

## Various Helper Routines

Below are various helper functions that can be useful when working with Entities. Notice how we use the Dynamic Query to load three columns. We do not load the employees ‘Address’ column.

### ContainsColumn and GetOriginalColumnValue

Look at the usage of two functions: **ContainsColumn** and **GetOriginalColumnValue**

EmployeesQuery query = new EmployeesQuery();

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

query.Where(query.EmployeeID == 2);

Employees employee = new Employees();

if (employee.Load(query))

{

if (employee.**ContainsColumn**(EmployeesMetadata.ColumnNames.Address))

{

throw new Exception("This column shouldn't exist");

}

employee.FirstName = "Samantha";

string orig =   
 (string)employee.**GetOriginalColumnValue**(EmployeesMetadata.ColumnNames.FirstName);

employee.Address = "100 Main Street";

if (!employee.**ContainsColumn**(EmployeesMetadata.ColumnNames.Address))

{

throw new Exception("This column should exist now !");

}

}

## Advanced Loading Techniques

This section contains examples of advanced loading techniques.

### Raw Text

This example uses a raw SQL string to populate the entity. 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 Employees : esEmployees

{

public Employees() { }

public bool CustomLoadWithRawText()

{

return this.Load(esQueryType.Text, "SELECT \* FROM [Employees] WHERE EmployeeID = 4");

}

}

Notice 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 an employee entity and call our custom method.

Employees employee = new Employees();

if (employee.CustomLoadWithRawText())

{

// Then we found our record

}

The Dynamic Query API can easily accomplish the above task and therefore should be used. However, the above example does show you how to use the protected Load() method.

### 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 Employees : esEmployees

{

public Employees() { }

public bool CustomLoadRawTextWithParams(int employeeIdMinimum, string lastNameLike)

{

return this.Load(esQueryType.Text,

"SELECT TOP 1 \* 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.

Employees employee = new Employees();

if (employee.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 an entity.

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

(

@EmployeeIdMinimum INT,

@LastNameLike NVARCHAR(20)

)

AS

BEGIN

SET NOCOUNT ON

DECLARE @Err int

SELECT TOP 1 \*

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 Employees : esEmployees

{

public Employees() { }

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.

Employees employee = new Employees();

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

{

// Then we found at least one record

}

## Other Protected Methods to Access Data

This section contains examples if special protected methods which can 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 Employees : esEmployees

{

public Employees() { }

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 executes SQL it doesn't load a entity 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.

Employees employee = new Employees();

DateTime? maxHireDate = employee.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 Employees : esEmployees

{

public Employees() { }

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.

Employees employee = new Employees();

int count = employee.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 Employees : esEmployees

{

public Employees() { }

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.

Employees employee = new Employees();

using (IDataReader rdr = coll.GetReaderForAllEmployees())

{

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

while (rdr.Read())

{

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

}

}