Entity Framework "Cheat Sheet"

Referencing Entity Framework

To use Entity Framework in your application you must reference it using the NuGet Package Manager.

Right-click on your project **References** folder and select **Manage NuGet Packages**.

In the Manage NuGet Packages dialog, enter "**Entity Framework**" into the **Search Online** box. Select Entity Framework from the search results and click the **Install** button. The required files will be downloaded and installed into your project.

Once installed you can reference the Entity Framework classes required in the remainder of this document.

To read more about NuGet, see http://www.nuget.org/ and http://docs.nuget.org/docs/start-here/Managing-NuGet-Packages-Using-The-Dialog.

Defining a Base Model

A base model allows us to define properties that every model is required to have. This will save creating a lot of redundant code.

A base model is simply a C# class that has the `abstract` modifier, indicating it can only be used as a base class (see http://msdn.microsoft.com/en-us/library/sf985hc5.aspx).

Id

Every entity framework model is **required** to have a unique `Id` that is the database key. The `Id` must have `[Key]` and `[Required]` attributes (see Data Annotations). The `Id` does not have to be called "`Id`".

Usually the `Id` is an `int`, however you can also use a `Guid` for a truly unique key. `Guid`s are recommended if you plan on synchronising multiple databases in the future.

For example:

```
[Key]
[Required]
public int Id { get; set; }
```

Other Properties

You can define any other common properties that you like. Model properties can be any standard C# data type, for example `string`, `int`, `DateTime`, `decimal`, `Guid`, etc.

For example we can define simple auditing properties to track when the model was created:

```
[Required]
[StringLength(100)]
public string CreatedBy { get; set; }

[Required]
public DateTime CreatedDate { get; set; }
```

Example Base Model

```
#region Using

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

#endregion

namespace SalesApp.Models
{
    /// <summary>
    /// The base model that contains all the core data for all other models.
    /// </summary>
    abstract class BaseModel
    {

        #region Id

        /// <summary>The unique database Id of the model.</summary>
        [Key]
        [Required]
        public int Id { get; set; }

        #endregion

        #region CreatedBy

        /// <summary>The user who created the model.</summary>
        [Required]
        [StringLength(100)]
        public string CreatedBy { get; set; }

        #endregion

        #region CreatedDate

        /// <summary>The date the model was created.</summary>
        [Required]
        public DateTime CreatedDate { get; set; }

        #endregion

        #region UpdatedBy

        /// <summary>The user who last updated the model.</summary>
        [Required]
        [StringLength(100)]
        public string UpdatedBy { get; set; }

        #endregion

        #region UpdatedDate

        /// <summary>The date the model was last updated.</summary>
        [Required]
        public DateTime UpdatedDate { get; set; }

        #endregion

    }
}
```

Creating a Model

Models are simple C# classes, also known as Plain Old CLR Objects (POCO). Because these are standard C# classes we can define any number of properties or methods in them.

Models should inherit from the Base Model so they get all the core properties and this also reduces the amount of common code in your models. For example:

```
public class SalesPerson : BaseModel
```

See http://msdn.microsoft.com/en-us/library/vstudio/dd456853(v=vs.100).aspx for more information on POCO.

Model properties can be any standard C# data type, for example `string`, `int`, `DateTime`, `decimal`, `Guid`, etc.

You can also use an enumeration and Entity Framework will store the enumeration value in the database and automatically map between the enumeration and the database. For example:

```
public System.DayOfWeek Day { get; set; }
```

Example Model

Note: This example uses the Base Model, `IActive` interface, `ObservableListSource` and Data Annotations.

```
#region Using

using SalesApp.Interfaces;

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

#endregion

namespace SalesApp.Models
{
    class SalesPerson : BaseModel, IActive
    {

        #region Active

        /// <summary>Flag indicating the sales person is active or inactive.</summary>
        [Required]
```

```
        public bool Active { get; set; }

        #endregion

        #region FirstName

        /// <summary>The first name of the sales person.</summary>
        [Required]
        [StringLength(100)]
        public string FirstName { get; set; }

        #endregion

        #region FullName

        /// <summary>The full name (first and last name) of the sales person.</summary>
        public string FullName
        {
            get
            {
                return string.Format("{0} {1}", FirstName, LastName).Trim();
            }
        }

        #endregion

        #region LastName

        /// <summary>The last name of the sales person.</summary>
        [Required]
        [StringLength(100)]
        public string LastName { get; set; }

        #endregion

        #region Region

        /// <summary>The sales region of the sales person.</summary>
        public virtual SalesRegion Region { get; set; }

        /// <summary>The database Id of the sales region of the sales person.</summary>
        [Required]
        public int RegionId { get; set; }

        #endregion

        #region Sales

        /// <summary>The sales made by the sales person.</summary>
        public virtual ObservableListSource<Sale> Sales { get; set; }

        #endregion

        #region SalesTarget

        /// <summary>The dollar sales target of the sales person.</summary>
        [Required]
        [Range(0, double.MaxValue)]
        public decimal SalesTarget { get; set; }

        #endregion

    }
}
```

Relationships

One-to-One

To map from one Model to another Model in a one-to-one mapping, simply add two properties to each Model, one for the opposite model and one for the opposite Models unique Id.

By following the naming convention of {model name}{id name} Entity Framework will automatically connect the `Id` property to the Model property.

For example:

```
#region Region

/// <summary>The sales region of the sales person.</summary>
public virtual SalesRegion Region { get; set; }

/// <summary>The database Id of the sales region of the sales person.</summary>
[Required]
public int RegionId { get; set; }

#endregion
```

One-to-Many

To map from one Model to many Models in a one-to-many mapping, simply add a one-to-one property on the first Model (see above), and on the second Model add a corresponding collection property (see Collections).

For example:

```
#region Sales

/// <summary>The sales made in this sales region.</summary>
public virtual ObservableListSource<Sale> Sales { get; set; }

#endregion
```

Data Annotations

Data Annotation attributes can be added to Model properties to tell Entity Framework how to treat the property. These are generally used for validation.

For example, a required `string` with maximum length of 100 characters:

```
[Required]
[StringLength(100)]
public string FirstName { get; set; }
```

A required `decimal` with minimum value of 0.0.

```
[Required]
[Range(0, double.MaxValue)]
public decimal SalesTarget { get; set; }
```

Note that many attributes have an Error Message parameter that can be used to provide a custom error message to users if validation fails.

For more details see:

Code First Data Annotations
http://msdn.microsoft.com/en-us/data/jj591583.aspx

System.ComponentModel.DataAnnotations Namespace
http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations(v=vs.110).aspx

Common Data Annotations

Attribute	Description
`[Column]`	Change the name of the database column for a property.
`[Key]`	The unique database key. For composite keys add [Key] to two or more properties.
`[MaxLength]`	The maximum length of a string.
`[MinLength]`	The minimum required length of a string.
`[NotMapped]`	Do not create a database column for the property.
`[Range]`	A numeric property must fit within the specified range.
`[Required]`	The property value is required (not null).
`[StringLength]`	A string property must fit within the specified range.
`[Table]`	Change the name of the database table for a model.

Collections

Collections of models should use `ICollection`, for example:

```
public virtual ICollection<Sale> Sales { get; set; }
```

However if you are creating a Windows Forms application you should use the following `ObservableListSource` class instead as it converts the collection into a `BindingList` more suitable to binding onto Windows Forms controls.

```
#region Using

using System;
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Data.Entity;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

#endregion

namespace SalesApp.Models
{
    /// <summary>
    /// A list that can be used for data binding on Windows Forms.
    /// For web applications use ICollection instead.
    /// </summary>
    /// <typeparam name="T">A model that derives from BaseModel.</typeparam>
    class ObservableListSource<T> : ObservableCollection<T>, IListSource where T : BaseModel
    {
        private IBindingList _bindingList;

        bool IListSource.ContainsListCollection { get { return false; } }

        IList IListSource.GetList()
        {
            return _bindingList ?? (_bindingList = this.ToBindingList());
        }
    }
}
```

Soft Deletion: IActive

By defining an `IActive` interface we can add a flag that determines if the model is active or inactive. Models can then be filtered based on `Active == true` (See CRUD: Read). See the Example Model for using the `IActive` interface in a model.

```
#region Using

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

#endregion

namespace SalesApp.Interfaces
{
    interface IActive
    {

        /// <summary>
        /// Flag indicating the entity is active or inactive.
        /// </summary>
        bool Active { get; set; }

    }
}
```

Data Context

The data context is the heart of the Entity Framework code.

By creating a class that inherits from the Entity Frameworks `DbContext` class we can create, read, update and delete any models in the database and control how Entity Framework creates the database structure. For example:

```
public class SalesContext : DbContext
```

DbSets

Each model in the database should be represented by a `DbSet` property on the data context. We create a simple property for each type of model:

```
public DbSet<Sale> Sales { get; set; }
```

OnModelCreating

We can also override the `OnModelCreating` method of the base `DbContext` class. This allows us to customize how the database is created and how Entity Framework treats the models, for example in the following code we remove the standard cascade delete behaviour:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
}
```

SaveChanges

The `DbContext` also has a `SaveChanges` method that can be overridden and customized. This method is called before any changes are written to the database. For example by customizing this method we can add functionality that performs soft deletion (see `IActive`) and auditing (see Base Model).

Example Data Context

```
#region Using

using SalesApp.Interfaces;
using SalesApp.Models;

using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.ModelConfiguration.Conventions;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

#endregion

namespace SalesApp.Data
{
    class SalesContext : DbContext
    {

        #region DbSets

        /// <summary>DbSet representing the Sales table.</summary>
        public DbSet<Sale> Sales { get; set; }

        /// <summary>DbSet representing the SalesPersons table.</summary>
        public DbSet<SalesPerson> People { get; set; }

        /// <summary>DbSet representing the SalesRegions table.</summary>
        public DbSet<SalesRegion> Regions { get; set; }

        #endregion

        #region OnModelCreating

        /// <summary>
        /// This method is called when the model for the context has been initialised, but
        /// before the model has been locked down and used to initialise the context.
        /// </summary>
        /// <param name="modelBuilder">
        /// The builder that defines the model for the context being created.
        /// </param>
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
        }

        #endregion

        #region SaveChanges

        /// <summary>
        /// Saves all changes made in this context to the underlying database.
        /// </summary>
        /// <returns>
        /// The number of objects written to the underlying database.
        /// </returns>
        public override int SaveChanges()
        {
            ChangeTracker.DetectChanges();

            // Get the state mananger to changed entities.
            var stateManager = ((IObjectContextAdapter)this).ObjectContext.ObjectStateManager;

            // "Soft" delete entities that implement IActive

            var deletedEntities = stateManager
                .GetObjectStateEntries(EntityState.Deleted)
```

```
                .Select(e => e.Entity)
                .OfType<IActive>()
                .ToArray();

            foreach (var deletedEntity in deletedEntities)
            {
                if (deletedEntity == null) continue;
                stateManager.ChangeObjectState(deletedEntity, EntityState.Modified);
                deletedEntity.Active = false;
            }

            // Auditing for newly created entities

            var createdEntities = stateManager
                .GetObjectStateEntries(EntityState.Added)
                .Select(e => e.Entity)
                .OfType<BaseModel>()
                .ToArray();

            foreach (var createdEntity in createdEntities)
            {
                createdEntity.CreatedDate = DateTime.Now;
                createdEntity.CreatedBy = Environment.UserName;
                createdEntity.UpdatedDate = DateTime.Now;
                createdEntity.UpdatedBy = Environment.UserName;
            }

            // Auditing for updated entities

            var updatedEntities = stateManager
                .GetObjectStateEntries(EntityState.Modified)
                .Select(e => e.Entity)
                .OfType<BaseModel>()
                .ToArray();

            foreach (var updatedEntity in updatedEntities)
            {
                updatedEntity.UpdatedDate = DateTime.Now;
                updatedEntity.UpdatedBy = Environment.UserName;
            }

            // Save change to the database
            return base.SaveChanges();
        }

        #endregion

    }
}
```

CRUD: Create

To create a new model, simply define a new variable of the correct Model type and add it to the appropriate Data Context `DbSet` using the `Add` method. You can add multiple new models using the `AddRange` method.

```
var person = new SalesPerson
{
    Active = true,
    FirstName = “Bob”,
    LastName = “Smith”,
    RegionId = 1,
    SalesTarget = 1000.00M
};

using (var context = new SalesContext())
{
    context.People.Add(person);
    var result = context.SaveChanges();

    MessageBox.Show(string.Format("{0} people created.", result));
}
```

CRUD: Read

Reading a list of models from the database:

```
using (var context = new SalesContext())
{
    var people = context.People
        .Where(p => p.Active)
        .OrderBy(p => p.FirstName)
        .ThenBy(p => p.LastName)
        .ToList();
}
```

Reading a filtered list of models from the database:

```
var personId = 1;
var regionId = 1;

using (var context = new SalesContext())
{
    var sales = context.Sales
        .Where(s => s.PersonId == personId &&
                    s.RegionId == regionId)
        .OrderBy(s => s.Date)
        .ToList();
}
```

Reading a single item from the database:

```
var personId = 1;

using (var context = new SalesContext())
{
    var person = context.People.SingleOrDefault(p => p.Id == personId);

    if (person != null)
    {
        MessageBox.Show(string.Format("{0} has a sales target of {1:C}",
            person.FullName, person.SalesTarget));
    }
}
```

Language Integrated Query (LINQ)

To learn more about LINQ queries, see:

LINQ Query Expressions (C# Programming Guide)
http://msdn.microsoft.com/en-us/library/bb397676.aspx

101 LINQ Samples
http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b

CRUD: Update

To update a model, simply change any properties as required and call SaveChanges on the Data Context. The Entity Framework will automatically track the changes and write them to the database after validating the data.

```
var salesId = 1;

using (var context = new SalesContext())
{
    var sale = context.Sales.SingleOrDefault(p => p.Id == salesId);

    if (sale != null)
    {
        sale.Amount = 231.56M;
        var result = context.SaveChanges();

        MessageBox.Show(string.Format("{0} sales updated.", result));
    }
}
```

CRUD: Delete

To delete a model from the database, appropriate Data Context `DbSet` using the `Add` method. Call `SaveChanges` to write the changes to the database.

If you have used the example `IActive` interface and Example Data Context then the Model will "soft delete" by having it's `Active` flag changed to false. The Model will not be permanently deleted from the database. Models can then be filtered based on `Active == true` (See CRUD: Read).

```
var salesId = 1;

using (var context = new SalesContext())
{
    var sale = context.Sales.SingleOrDefault(p => p.Id == salesId);

    if (sale != null)
    {
        context.Sales.Remove(sale);
        var result = context.SaveChanges();

        MessageBox.Show(string.Format("{0} sales deleted.", result));
    }
}
```

Calling Custom SQL Code

You can call custom SQL Server code from within your Data Context by using the `Database.SqlQuery` method.

For example, in the following code a stored procedure called `SalesReport` is called and passed two date parameters for the start and end period of the report.

The stored procedure would then return database rows which are converted by Entity Framework into a list of `SalesReportItems` (not shown). This Model would have properties that map to the columns returned by the stored procedure.

```
using System.Data.SqlClient;

...

#region GetSalesReport

public IList<SalesReportItem> GetSalesReport(DateTime startDate, DateTime endDate)
{
    SqlParameter startParam = new SqlParameter("@StartDate", startDate);
    SqlParameter endParam = new SqlParameter("@EndDate", endDate);
    return this.Database.SqlQuery<SalesReportItem>(
        "SalesReport @StartDate, @EndDate", startParam, endParam).ToList();
}

#endregion
```