Magnus Montin

.NET

My blog about application development on the .NET platform and Windows®.

Implementing a generic data access layer using Entity Framework

Posted: May 30, 2013 | **Author:** Magnus Montin | **Filed under:** Entity Framework, N-tier | **Tags:** .NET, C#, Entity Framework, N-tier |141 Comments
This post is about how you can develop a generic data access layer (DAL) with full CRUD (Create, Read, Update and Delete) support using Entity Framework 5 with plain old CLR objects (POCOs) and short-lived contexts in a disconnected and stateless N-tier application.

Entity Framework (EF) is Microsoft's recommended data access technology when building new .NET applications. It is an object-relational mapping framework (ORM) that enables developers to work with relational data using domain-specific objects without having to write code to access data from a database.

EF provides three options for creating the conceptual model of your domain entities, also known as the entity data model (EDM); database first, model first and code first. With both the model first and code first approaches the presumption is that you don't have an existing database when you start developing the application and a database schema is created based on the model. As databases within enterprise environments are generally designed and maintained by database administrators (DBAs) rather than developers, this post will use the database first option where the EDM becomes a virtual reflection of a database or a subset of it.

Typically when you are doing database first development using EF you are targeting an already existing database but for testing and demo purposes you may of course generate a new one from scratch. There is a walkthrough on how you can create a local service-based database in Visual Studio 2012 available on MSDN here.

The database used in this example is a very simple one containing only the two tables listed below. There is a one-to-many relationship between the *Department* and *Employee* tables meaning an employee belongs to a single department and a department can have several employees.

```
CREATE TABLE [dbo].[Department] (
    [DepartmentId] INT          IDENTITY (1, 1) NOT NULL,
    [Name]         VARCHAR (50) NULL,
    PRIMARY KEY CLUSTERED ([DepartmentId] ASC)
);

CREATE TABLE [dbo].[Employee] (
    [EmployeeId]   INT          IDENTITY (1, 1) NOT NULL,
    [DepartmentId] INT          NOT NULL,
    [FirstName]    VARCHAR (20) NOT NULL,
    [LastName]     VARCHAR (20) NOT NULL,
    [Email]        VARCHAR (50) NULL,
    PRIMARY KEY CLUSTERED ([EmployeeId] ASC),
    CONSTRAINT [FK_Employee_Department] FOREIGN KEY ([DepartmentId])
    REFERENCES [dbo].[Department] ([DepartmentId])
);
```

N-tier architecture

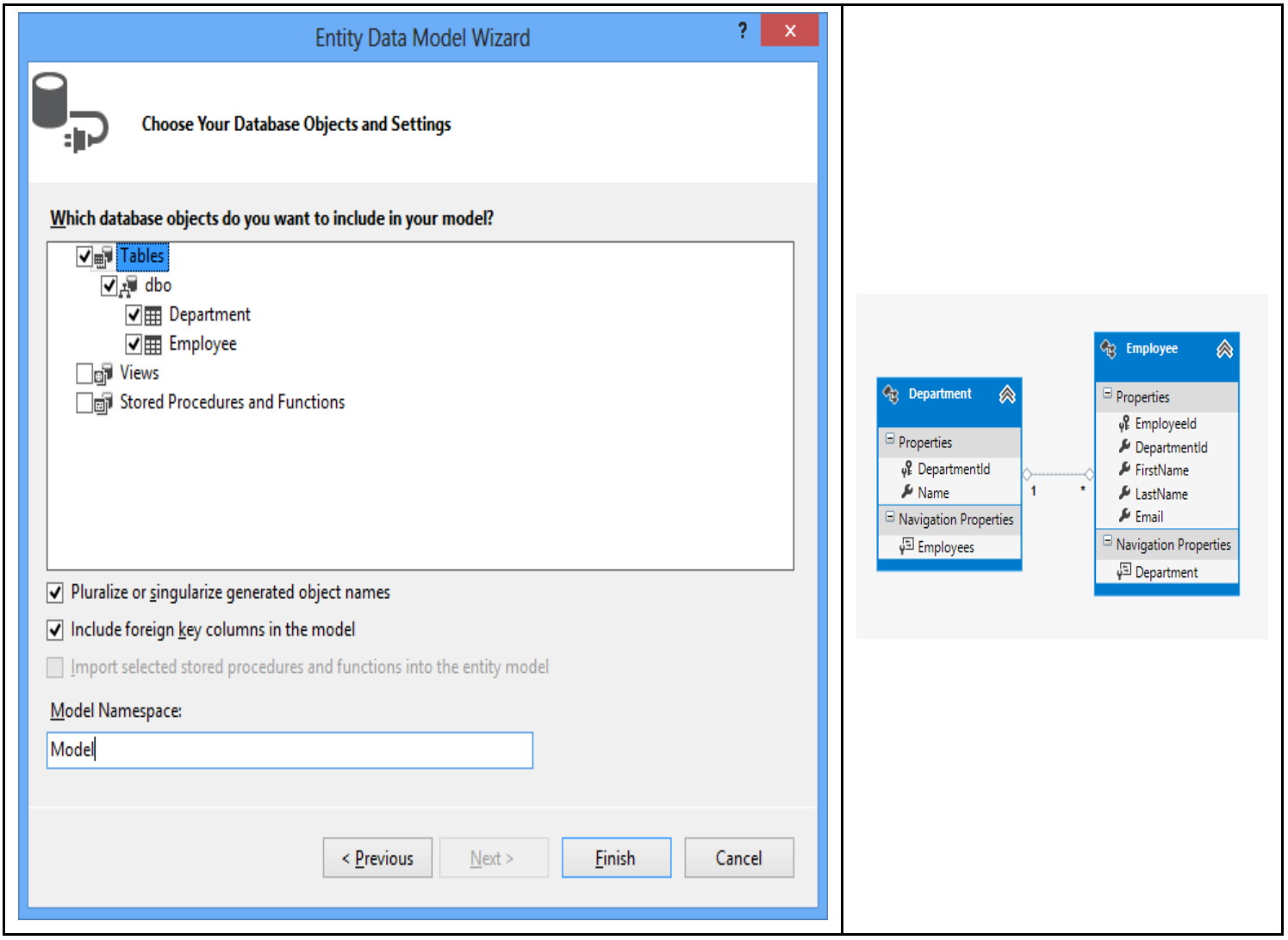
A large enterprise application will typically have one or more databases to store data and on top of this a data access layer (DAL) to access the database(s). On top of this there may be some repositories to communicate with the DAL, a business layer containing logic and classes representing the business domain, a service layer to expose the business layer to clients and finally some user interface application such as a WPF desktop application or an ASP.NET web application.

User interface layer WPF / ASP.NET / Console App / WinRT / …
Service layer WCF / ASMX / …
Business logic layer
Data access layer EF / ADO.NET / …
Database SQL Server / Oracle / MySql / …

Data access layer (DAL)

The DAL is simply a C# class library project where you define the model generated from the existing database along with the generic implementation for reading and modifying the data in the database. It is the only layer in the application that will actually know anything about and have any dependencies on EF. Any user interface code should only communicate with the service or business layer and don't have any references to the DAL.

1. Start by creating a new class library project (*Mm.DataAccessLayer*) and add a new **ADO.NET Entity Data Model** to it. Choose the "Generate from database" option in the Entity Data Model wizard. The wizard lets you connect to the database and select the *Department* and *Employee* tables to be included in the model.



Once the wizard has completed the model is added to your project and you are able to view it in the EF Designer. By default all generated code including the model classes for the *Department* and *Employee* entities sits in the same project.

Separating entity classes from EDMX

Again, in an enterprise level application where separation of concerns is of great importance you certainly want to have your domain logic and your data access logic in separate projects. In other words you want to move the generated model (Model.tt) to another project. This can easily be accomplished by following these steps:

2. Add a new class library project (*Mm.DomainModel*) to the solution in Visual Studio.
3. Open File Explorer (right-click on the solution in Visual Studio and choose the "Open Folder in File Explorer" option) and move the Model.tt file to the new project folder.
4. Back in Visual Studio, include the Model.tt file in the new project by clicking on the "Show All Files" icon at the top of the Solution Explorer and then right-click on the Model.tt file and choose the "Include In Project" option.
5. Delete the Model.tt file from the DAL project.
6. For the template in the new domain model project to be able to find the model you then need to modify it to point to the correct EDMX path. You do this by setting the inputFile variable in the Model.tt template file to point to an explicit path where to find the model:

```
const string inputFile = @"../Mm.DataAccessLayer/Model.edmx";
```

Once you save the file the entity classes should be generated in the domain model project. Note that if you make any changes to the model in the DAL project later on you are required to explicitly update your model classes. By right-click on the Model.tt template file and choose "Run Custom Tool" the entity classes will be regenerated to reflect the latest changes to the model.

7. As the context by default expects the entity classes to be in the same namespace, add a using statement for their new namespace to the Model.Context.tt template file in the DAL project:

```
using System;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using Mm.DomainModel; <!-- Added -->
<#
if (container.FunctionImports.Any())
{
#>
using System.Data.Objects;
using System.Data.Objects.DataClasses;
using System.Linq;
<#
}
#>
```

8. Finally, you need to add a reference from the DAL project to the domain model project in order for it to compile.

DbContext

In an EF-based application a context is responsible for tracking changes that are made to the entities after they have been loaded from the database. You then use the SaveChanges method on the context to persist the changes back to the database.

By default EDMs created in Visual Studio 2012 generates simple POCO entity classes and a context that derives from *DbContext* and this is the recommended template unless you have a reason to use one of the others listed on MSDN here.

The *DbContext* class was introduced in EF 4.1 and provides a simpler and more lightweight API compared to the EF 4.0 *ObjectContext*. However it simply acts like a wrapper around the *ObjectContext* and if you for some reason need the granular control of the latter you can implement an extension method – extension methods enable you to "add" methods to existing types without creating a new derived type – to be able to convert the *DbContext* to an *ObjectContext* through an adapter:

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.Objects;

namespace Mm.DataAccessLayer
{
    public static class DbContextExtensions
    {
        public static ObjectContext ToObjectContext(this DbContext dbContext)
        {
            return (dbContext as IObjectContextAdapter).ObjectContext;
        }
    }
}
```

Encapsulating data access into repositories

A repository is responsible for encapsulating the data access code. It sits between the DAL and the business layer of the application to query the data source for data and map this data to an entity class, and it also persists changes in the entity classes back to the data source using the context.

A repository typically implements an interface that provides a simple set of methods for the developer using the repository to code against. Using an interface the consumer doesn't need to know anything about what happens behind the scenes, i.e. whether the DAL uses EF, another ORM or manually creating connections and commands to execute queries against a data source. Besides the abstraction it brings it's also great if you are using dependency injection in your application.

By using a generic repository for querying and persisting changes for your entity classes you can maximize code reuse. Below is a sample generic interface which provides methods to query for all entities, specific entities matching a given where predicate and a single entity as well as methods for inserting, updating and removing an arbitrary number of entities.

9. Add the below interface named *IGenericDataRepository* to the *Mm.DataAccessLayer* project.

```
using Mm.DomainModel;
using System;
using System.Collections.Generic;
using System.Linq.Expressions;

namespace Mm.DataAccessLayer
{
    public interface IGenericDataRepository<T> where T : class
    {
        IList<T> GetAll(params Expression<Func<T, object>>[] navigationProperties);
        IList<T> GetList(Func<T, bool> where, params Expression<Func<T, object>>[] na
        T GetSingle(Func<T, bool> where, params Expression<Func<T, object>>[] navigat
        void Add(params T[] items);
        void Update(params T[] items);
        void Remove(params T[] items);
    }
}
```

IList vs IQueryable

Note that the return type of the two Get* methods is *IList<T>* rather than *IQueryable<T>*. This means that the methods will be returning the actual already executed results from the queries rather than executable queries themselves. Creating queries and return these back to the calling code would make the caller responsible for executing the LINQ-to-Entities queries and consequently use EF logic. Besides, when using EF in an N-tier application the repository typically creates a new context and dispose it on every request meaning the calling code won't have access to it and therefore the ability to cause the query to be executed. Thus you should always keep your LINQ queries inside of the repository when using EF in a disconnected scenario such as in an N-tier application.

Loading related entities

EF offers two categories for loading entities that are related to your target entity, e.g. getting employees associated with a department in this case. Eager loading uses the Include method on the *DbSet* to load child entities and will issue a single query that fetches the data for all the included entities in a single call. Each of the methods for reading data from the database in the concrete sample implementation of the *IGenericDataRepository<T>* interface below supports eager loading by accepting a variable number of navigation properties to be included in the query as arguments.

10. Add a new class named *GenericDataRepository* to the *MM.DataAccessLayer* project and implement the *IGenericDataRepository<T>* interface .

```
using Mm.DomainModel;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Linq.Expressions;

namespace Mm.DataAccessLayer
{
    public class GenericDataRepository<T> : IGenericDataRepository<T> where T : class
    {
        public virtual IList<T> GetAll(params Expression<Func<T, object>>[] navigati
        {
            List<T> list;
            using (var context = new Entities())
            {
                IQueryable<T> dbQuery = context.Set<T>();

                //Apply eager loading
                foreach (Expression<Func<T, object>> navigationProperty in navigatior
                    dbQuery = dbQuery.Include<T, object>(navigationProperty);

                list = dbQuery
                    .AsNoTracking()
                    .ToList<T>();
            }
            return list;
        }
```

```
        public virtual IList<T> GetList(Func<T, bool> where,
             params Expression<Func<T,object>>[] navigationProperties)
        {
            List<T> list;
            using (var context = new Entities())
            {
                IQueryable<T> dbQuery = context.Set<T>();

                //Apply eager loading
                foreach (Expression<Func<T, object>> navigationProperty in navigatior
                    dbQuery = dbQuery.Include<T, object>(navigationProperty);

                list = dbQuery
                    .AsNoTracking()
                    .Where(where)
                    .ToList<T>();
            }
            return list;
        }

        public virtual T GetSingle(Func<T, bool> where,
             params Expression<Func<T, object>>[] navigationProperties)
        {
            T item = null;
            using (var context = new Entities())
            {
                IQueryable<T> dbQuery = context.Set<T>();

                //Apply eager loading
                foreach (Expression<Func<T, object>> navigationProperty in navigatior
                    dbQuery = dbQuery.Include<T, object>(navigationProperty);

                item = dbQuery
                    .AsNoTracking() //Don't track any changes for the selected item
                    .FirstOrDefault(where); //Apply where clause
            }
            return item;
        }

        /* rest of code omitted */
    }
}
```

For example, here's how you would call the GetAll method to get all departments with its employees included:

```
IGenericDataRepository<Department> repository = new GenericDataRepository<Department>
IList<Department> departments = repository.GetAll(d => d.Employees);
```

With lazy loading related entities are loaded from the data source by EF issuing a separate query first when the get accessor of a navigation property is accessed programmatically.

Dynamic proxies

For EF to enable features such as lazy loading and automatic change tracking for POCO entities, it can create a wrapper class around the POCO entity at runtime. There is a certain set of rules that your entity classes need to follow to get this proxy behavior. To get the instant change tracking behavior every property must be marked as virtual. For the lazy loading to work, those related properties that you want to be lazily loaded must be marked as virtual and those who point to a set of related child objects have to be of type ICollection. There is a complete list of the requirements for POCO proxies to be created available on MSDN here if you want more information.

Disconnected entities

However, in an N-tier application entity objects are usually disconnected meaning they are not being tracked by a context as the data is fetched using one context, returned to the client where there is no context to track changes and then sent back to the server and persisted back to the database using another instance of the context. Looking at the code above, a new instance of the context will be created and disposed for each method call and the AsNoTracking extension method – also added in EF 4.1 – is used to tell the context not to track any changes which may result in better performance when querying for a large number of entities. When using short-lived contexts like this, you should disable lazy loading. If you don't an exception saying the context has been disposed will be thrown whenever a non-initialized navigation property is accessed from anywhere outside the context's scope.

11. Lazy loading and dynamic proxy creation is turned off for all entities in a context by setting two flags on the Configuration property on the *DbContext* as shown below. Both these properties are set to true by default.

```
namespace Mm.DataAccessLayer
{
    using System.Data.Entity;
    using System.Data.Entity.Infrastructure;
    using Mm.DomainModel;

    public partial class Entities : DbContext
    {
        public Entities()
            : base("name=Entities")
        {
            Configuration.LazyLoadingEnabled = false;
            Configuration.ProxyCreationEnabled = false;
        }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            throw new UnintentionalCodeFirstException();
        }

        public DbSet<Department> Departments { get; set; }
        public DbSet<Employee> Employees { get; set; }
    }
}
```

Root vs Graphs

When it comes to persisting changes to the database you need to decide whether your CUD methods should accept an entire graph of entities or only a single root entity to be passed in. A graph of entities is a number of entities that reference each other. For example, when you want to insert a new *Department* entity to the database by passing it to the repository's Add method it might have related *Employee* objects. In this case the *Employee* objects belong to the graph and the *Department* object is the root entity.

EntityState

On the server side, things will get easier if you decide to not support graphs. In this case you could expose an Add method and an Update method for each entity type and these methods would only operate on a standalone instance rather than a graph of entities. EF makes it simple to implement these methods. It is all about setting the state of the passed in entity object. An entity can be in one of five states as defined by the *System.Data.EntityState* enumeration:

–*Added*: the entity is being tracked by the context but hasn't been added to the database yet.

–*Unchanged*: the entity is being tracked by the context, it exists in the database but its property values have not been changed since it was fetched from the database.

–*Modified*: the entity is being tracked by the context, it exists in the database and some or all of its property values have been modified since it was fetched from the database

–*Deleted*: the entity is being tracked by the context, it exists in the database but will be deleted on the next call to the SaveChanges method.

–*Detached*: the entity is not being tracked by the context at all.

When the context's SaveChanges method is called it decides what to do based on the entity's current state. Unchanged and detached entities are ignored while added entities are inserted into the database and then become *Unchanged* when the method returns, modified entities are updated in the database and then become *Unchanged* and deleted entities are deleted from the database and then detached from the context.

DbSet.Entry

You can explicitly change the state of an entity by using the DbSet.Entry method. There is no need to attach the entity to the context before using this method as it will automatically do the attachment if needed. Below is the implementation of the generic repository's Add method. It explicitly sets the state of the entity to be inserted into the database to *Added* before calling SaveChanges to execute and commit the insert statement.

```
public virtual void Add(params T[] items)
{
    using (var context = new Entities())
    {
        foreach (T item in items)
        {
            context.Entry(item).State = System.Data.EntityState.Added;
        }
        context.SaveChanges();
    }
}
```

The implementation for the Update and Remove methods are very similar to the Add method as shown below. Note that all exception handling has been omitted for brevity in the sample code.

```
public virtual void Update(params T[] items)
{
    using (var context = new Entities())
    {
        foreach (T item in items)
        {
            context.Entry(item).State = System.Data.EntityState.Modified;
        }
        context.SaveChanges();
    }
}

public virtual void Remove(params T[] items)
{
    using (var context = new Entities())
    {
        foreach (T item in items)
        {
            context.Entry(item).State = System.Data.EntityState.Deleted;
        }
        context.SaveChanges();
    }
}
```

Also note that all methods have been marked as virtual. This allows you to override any method in the generic repository by adding a derived class in cases where you need some specific logic to apply only to a certain type of entity. To be able to extend the generic implementation with methods that are specific only to a certain type of entity, whether it's an initial requirement or a possible future one, it's considered a good practice to define a repository per entity type from the beginning. You can simply inherit these repositories from the generic one as shown below and add methods to extend the common functionality based on your needs.

12. Add interfaces and classes to represent specific repositories for the *Department* and *Employee* entities to the DAL project.

```
using Mm.DomainModel;

namespace Mm.DataAccessLayer
{
    public interface IDepartmentRepository : IGenericDataRepository<Department>
    {
    }

    public interface IEmployeeRepository : IGenericDataRepository<Employee>
    {
    }

    public class DepartmentRepository : GenericDataRepository<Department>, IDepartmer
    {
    }

    public class EmployeeRepository : GenericDataRepository<Employee>, IEmployeeRepos
    {
    }
}
```

Business layer

As mentioned before, the repository is located somewhere between the DAL and the business layer in a typical N-tier architecture. The business layer will use it to communicate with the database through the EDM in the DAL. Any client application will be happily unaware of any details regarding how data is fetched or persisted on the server side. It's the responsibility of the business layer to provide methods for the client to use to communicate with the server.

13. Add a new project (*Mm.BusinessLayer*) to the solution with references to the DAL project (*Mm.DataAccessLayer*) and the project with the domain classes (*Mm.DomainModel*). Then add a new interface and a class implementing this interface to it to expose methods for creating, reading, updating and deleting entities to any client application.

Below is a sample implementation. In a real world application the methods in the business layer would probably contain code to validate the entities before processing them and it would also be catching and logging exceptions and maybe do some caching of frequently used data as well.

```
using Mm.DomainModel;
using System.Collections.Generic;
using Mm.DataAccessLayer;

namespace Mm.BusinessLayer
{
    public interface IBusinessLayer
    {
        IList<Department> GetAllDepartments();
        Department GetDepartmentByName(string departmentName);
        void AddDepartment(params Department[] departments);
        void UpdateDepartment(params Department[] departments);
        void RemoveDepartment(params Department[] departments);

        IList<Employee> GetEmployeesByDepartmentName(string departmentName);
```

```
        void AddEmployee(Employee employee);
        void UpdateEmploee(Employee employee);
        void RemoveEmployee(Employee employee);
    }

    public class BuinessLayer : IBusinessLayer
    {
        private readonly IDepartmentRepository _deptRepository;
        private readonly IEmployeeRepository _employeeRepository;

        public BuinessLayer()
        {
            _deptRepository = new DepartmentRepository();
            _employeeRepository = new EmployeeRepository();
        }

        public BuinessLayer(IDepartmentRepository deptRepository,
            IEmployeeRepository employeeRepository)
        {
            _deptRepository = deptRepository;
            _employeeRepository = employeeRepository;
        }

        public IList<Department> GetAllDepartments()
        {
            return _deptRepository.GetAll();
        }

        public Department GetDepartmentByName(string departmentName)
        {
            return _deptRepository.GetSingle(
                d => d.Name.Equals(departmentName),
                d => d.Employees); //include related employees
        }

        public void AddDepartment(params Department[] departments)
        {
            /* Validation and error handling omitted */
            _deptRepository.Add(departments);
        }

        public void UpdateDepartment(params Department[] departments)
        {
            /* Validation and error handling omitted */
            _deptRepository.Update(departments);
        }

        public void RemoveDepartment(params Department[] departments)
        {
            /* Validation and error handling omitted */
            _deptRepository.Remove(departments);
        }

        public IList<Employee> GetEmployeesByDepartmentName(string departmentName)
        {
            return _employeeRepository.GetList(e => e.Department.Name.Equals(departme
        }
```

```
        public void AddEmployee(Employee employee)
        {
            /* Validation and error handling omitted */
            _employeeRepository.Add(employee);
        }

        public void UpdateEmploee(Employee employee)
        {
            /* Validation and error handling omitted */
            _employeeRepository.Update(employee);
        }

        public void RemoveEmployee(Employee employee)
        {
            /* Validation and error handling omitted */
            _employeeRepository.Remove(employee);
        }
    }
}
```

Client

A client application consuming the sever side code will only need references to the business layer and the entity classes defined in the *Mm.DomainModel* project. Below is a simple C# console application to test the functionality provided by the business layer. It's important to note that there are no references or dependencies to EF in this application. In fact you could replace the EF-based DAL with another one using raw T-SQL commands to communicate with the database without affecting the client side code. The only thing in the console application that hints that EF may be involved is the connection string that was generated in the DAL project when the EDM was created and has to be added to the application's configuration file (App.config). Connection strings used by EF contain information about the required model, the mapping files between the model and the database and how to connect to the database using the underlying data provider.

14. To be able to test the functionality of the business layer and the DAL, create a new console application and add references to the *Mm.BusinessLayer* project and the *Mm.DomainModel* project.

```
using Mm.BusinessLayer;
using Mm.DomainModel;
using System;
using System.Collections.Generic;

namespace Mm.ConsoleClientApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            IBusinessLayer businessLayer = new BuinessLayer();

            /* Create some departments and insert them to the database through the bu
            Department it = new Department() { Name = "IT" };
            Department sales = new Department() { Name = "Sales" };
```

```
            Department marketing = new Department() { Name = "Marketing" };
            businessLayer.AddDepartment(it, sales, marketing);

            /* Get a list of departments from the database through the business layer
            Console.WriteLine("Existing departments:");
            IList<Department> departments = businessLayer.GetAllDepartments();
            foreach (Department department in departments)
                Console.WriteLine(string.Format("{0} - {1}", department.DepartmentId,


            /* Add a new employee and assign it to a department */
            Employee employee = new Employee()
            {
                FirstName = "Magnus",
                LastName = "Montin",
                DepartmentId = it.DepartmentId
            };
            businessLayer.AddEmployee(employee);

            /* Get a single department by name */
            it = businessLayer.GetDepartmentByName("IT");
            if (it != null)
            {
                Console.WriteLine(string.Format("Employees at the {0} department:", i
                foreach (Employee e in it.Employees)
                    Console.WriteLine(string.Format("{0}, {1}", e.LastName, e.FirstNa
            };

            /* Update an existing department */
            it.Name = "IT Department";
            businessLayer.UpdateDepartment(it);

            /* Remove employee */
            it.Employees.Clear();
            businessLayer.RemoveEmployee(employee);

            /* Remove departments*/
            businessLayer.RemoveDepartment(it, sales, marketing);

            Console.ReadLine();
        }
    }
}
```

Persisting disconnected graphs

While avoiding the complexity of accepting graphs of objects to be persisted at once makes life easier for server side developers, it potentially makes the client component more complex. As you may have noticed by looking at the code for the business layer above, you are also likely to end up with a large number of operations exposed from the server. If you do want your business layer to be able to handle graphs of objects to passed in and be persisted correctly, you need a way of determining what changes were made to the passed in entity objects in order for you to set their states correctly.

For example, consider a scenario when you get a *Department* object representing a graph with related *Employee* objects. If all entities in the graph are new, i.e. are not yet in the database, you can simply call the DbSet.Add method to set the state of all entities in the graph to *Added* and call the SaveChanges to persist the changes. If the root entity, the *Department* in this case, is new and all related *Employee* objects are unchanged and already existing in the database you can use the DbSet.Entry method to change the state of the root only. If the root entity is modified and some related items have also been changed, you would first use the DbSet.Entry method to set the state of the root entity to *Modified*. This will attach the entire graph to the context and set the state of the related objects to *Unchanged*. You will then need to identify the related entities that have been changed and set the state of these to *Modified* too. Finally, you may have a graph with entities of varying states including added ones. The best thing here is to use the DbSet.Add method to set the states of the related entities that were truly added to *Added* and then use the DbSet.Entry method to set the correct state of the other ones.

So how do you know the state of an entity when it comes from a disconnected source and how do you make your business layer able to persist a graph with a variety of objects with a variety of states? The key here is to have the entity objects track their own state by explicitly setting the state on the client side before passing them to the business layer. This can be accomplished by letting all entity classes implement an interface with a state property. Below is a sample interface and an enum defining the possible states.

```
namespace Mm.DomainModel
{
    public interface IEntity
    {
        EntityState EntityState { get; set; }
    }

    public enum EntityState
    {
        Unchanged,
        Added,
        Modified,
        Deleted
    }
}
```

```
/* Entity classes implementing IEntity */
public partial class Department : IEntity
{
    public Department()
    {
        this.Employees = new HashSet<Employee>();
    }

    public int DepartmentId { get; set; }
    public string Name { get; set; }

    public virtual ICollection<Employee> Employees { get; set; }

    public EntityState EntityState { get; set; }
}

public partial class Employee : IEntity
{
    public int EmployeeId { get; set; }
    public int DepartmentId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }

    public virtual Department Department { get; set; }

    public EntityState EntityState { get; set; }
}
```

With this solution, the business layer will know the state of each entity in a passed in graph assuming the states have been set correctly in the client application. The repository will need a helper method to convert the custom *EntityState* value to a *System.Data.EntityState* enumeration value. The below static method can be added to the *GenericDataRepository<T>* class in the DAL to takes care of this.

```
protected static System.Data.EntityState GetEntityState(Mm.DomainModel.EntityState er
{
    switch (entityState)
    {
        case DomainModel.EntityState.Unchanged:
            return System.Data.EntityState.Unchanged;
        case DomainModel.EntityState.Added:
            return System.Data.EntityState.Added;
        case DomainModel.EntityState.Modified:
            return System.Data.EntityState.Modified;
        case DomainModel.EntityState.Deleted:
            return System.Data.EntityState.Deleted;
        default:
            return System.Data.EntityState.Detached;
    }
}
```

Next, you need to specify a constraint on the *IGenericDataRepository<T>* interface and the *GenericDataRepository<T>* class to ensure that the type parameter T implements the *IEntity* interface and then make some modifications to the CUD methods in the repository as per below. Note that the Update

method will actually be able to do all the work now as it basically only sets the *System.Data.EntityState* of an entity based on the value of the custom enum property.

```
public interface IGenericDataRepository<T> where T : class, IEntity { ... }

public virtual void Add(params T[] items)
{
    Update(items);
}

public virtual void Update(params T[] items)
{
    using (var context = new Entities())
    {
        DbSet<T> dbSet = context.Set<T>();
        foreach (T item in items)
        {
            dbSet.Add(item);
            foreach (DbEntityEntry<IEntity> entry in context.ChangeTracker.Entries<IE
            {
                IEntity entity = entry.Entity;
                entry.State = GetEntityState(entity.EntityState);
            }
        }
        context.SaveChanges();
    }
}

public virtual void Remove(params T[] items)
{
    Update(items);
}
```

Also note they key to all this working is that the client application must set the correct state of an entity as the repository will be totally dependent on this. Finally, below is some client side code that shows how to set the state of entities and passing a graph of objects to the business layer.

```
using Mm.BusinessLayer;
using Mm.DomainModel;
using System;
using System.Collections.Generic;

namespace Mm.ConsoleClientApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            IBusinessLayer businessLayer = new BuinessLayer();

            /* Create a department graph with two related employee objects */
            Department it = new Department() { Name = "IT" };
            it.Employees = new List<Employee>
            {
                new Employee { FirstName="Donald", LastName="Duck", EntityState=Entit
                new Employee { FirstName="Mickey", LastName="Mouse", EntityState=Ent
```

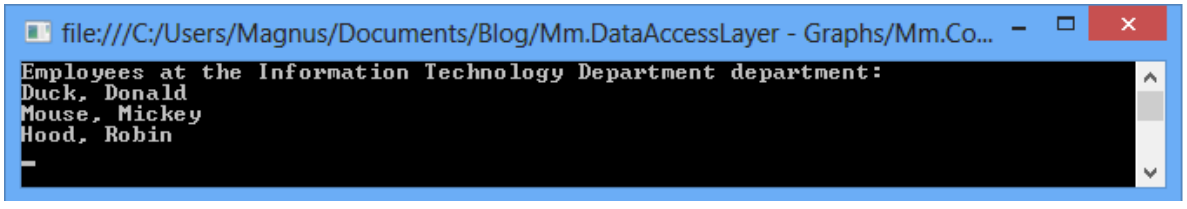
```
            };
            it.EntityState = EntityState.Added;
            businessLayer.AddDepartment(it);

            /*Add another employee to the IT department */
            Employee employee = new Employee()
            {
                FirstName = "Robin",
                LastName = "Hood",
                DepartmentId = it.DepartmentId,
                EntityState = EntityState.Added
            };
            /* and change the name of the department */
            it.Name = "Information Technology Department";
            it.EntityState = EntityState.Modified;
            foreach (Employee emp in it.Employees)
                emp.EntityState = EntityState.Unchanged;
            it.Employees.Add(employee);
            businessLayer.UpdateDepartment(it);

            /* Verify changes by quering for the updated department */
            it = businessLayer.GetDepartmentByName("Information Technology Departmen
            if (it != null)
            {
                Console.WriteLine(string.Format("Employees at the {0} department:", 
                foreach (Employee e in it.Employees)
                    Console.WriteLine(string.Format("{0}, {1}", e.LastName, e.FirstNa
            };

            /* Delete all entities */
            it.EntityState = EntityState.Deleted;
            foreach (Employee e in it.Employees)
                e.EntityState = EntityState.Deleted;
            businessLayer.RemoveDepartment(it);

            Console.ReadLine();
        }
    }
}
```



141 Comments on “Implementing a generic data access layer using Entity Framework”

1. Ricardo Aranibar says:

October 17, 2014 at 15:35
If you lose IEntity in your model every time it is regenerated.
you should change in properties Model.Contex.tt the namespace to entities.

2. Alien says:
October 29, 2014 at 11:34
Superb article :) Though it took me some time to get it fully working for me.

One thing I'm wondering about is how to get just some columns from a an entity or a join of entities?
This is so WCF won't have to get all that data and then filter it on the client side.
Some queries would get very big with lots of data not needed.
What would be the best practice when using your sample.

3. Magnus Montin says:
October 29, 2014 at 18:02
Hi Alien,

Only load the related entities of an entity that are necassary or you could return Data Transfer Objects (DTOs) that contains only a subset of the properties of an actual entity class from the business layer instead of returning the actual entity object(s). DTOs are out of the scope of this article though.

4. Alien says:
October 30, 2014 at 19:08
thx for the reply

I was trying to skip using DTOs to not create more maintenance of extra objects than I need.

I got big problems loading just those entities that I need.
I never get that 'lamda' expression working, either it's some syntax errors in it or the result ain't what I want.

Think you could give an example of at least 2 columns and I should be able to figure out the rest.

5. Alien says:
November 4, 2014 at 17:59
Hi again.

Just wonder if you have a solution for the IntelliSence for a VB.NET project using your implementation?
I just started with one and the functions trying to go with IntelliSense to it's parents/children donsn't work.
It works if you spell it correctly but if it's easy to fix I'd rather use that :)

6. TPhelps says:
November 6, 2014 at 03:36
My organization's standards require a call to a stored procedure to insert a row; the stored procedure returns the inserted row id. I am unsure what changes would need to be made to the code to return a list of row ids corresponding to the items parameter that were added. Can you provide?

Thanks in advance

7. Vladimir Venegas says:
   November 18, 2014 at 17:49
   Hi, this very good tutorial, I have used as the basis for doing something similar. I have a problem that maybe you can help me.

   The following lines of code work fine:

   ```
   IQueryable query = _dataContext.Set();
   query.Where(_where);
   return query.AsNoTracking().Include(_include[0]).ToList();
   ```

   But the following lines of code do not bring the include:

   ```
   IQueryable query = _dataContext.Set();
   query.Where(_where);
   foreach (var include in _include)
   {
   query.Include(include);
   }
   return query.AsNoTracking().ToList();
   ```

   This is the class constructor:

   ```
   //Class constructor
   public Repository(DbContext dataContext)
   {
   _dataContext = dataContext;
   _dataContext.Configuration.LazyLoadingEnabled = false;
   _dataContext.Configuration.ProxyCreationEnabled = false;
   }
   ```

8. Alexandre says:
   November 20, 2014 at 17:31
   Hi Magnus,

   I like your example. The only problem I had with it was the Func where instead of Expression, which makes the filtering and the select top 1 occur after retrieving everything. I see you talked about it in the comments already but unless someone checks the comments or the sql profiler, they might never know what really happens.

   Thanks anyways, it helped me greatly!

9. Alien says:
   November 24, 2014 at 11:17
   Another thing that I noticed is when trying to update Many-to-Many Entities using this code.
   No matter what EntityState I use it won't work. It seems to always want to insert rows in the related tables instead of only the link table creating a Duplicate Error.
   Have I missed something or can't this be done using this approach?

   Anyone got a solution for this?

10. Julien says:
    November 25, 2014 at 12:05
    @Alien

    I use this package to update many-to-many entities :
    https://github.com/refactorthis/GraphDiff

    If anyone has a better solution… Tell us about it !

11. Alien says:
    December 5, 2014 at 16:57
    @Julien, thx I'll take a look at that later

    Another thing I just came across: "AcceptChanges cannot continue because the object's key values conflict with another object in the ObjectStateManager"

    I've read a lot about this and many seems to leave ORM's just because of this.
    The issue is that I'm assigning the same Segment key twice in the context, which mucks up the ObjectStateManager.

    For me this is a major bug in EF, What entity graphs don't have same object referenced to different entities?
    eg. a Person entity has a reference to Address.
    You load all persons with related data for some eventual changes and then try to save it.
    That graph will contain SAME Segment key for persons living at the same address thus giving this error when trying to:
    'entry.State = GetEntityState(entity.EntityState);' as shown in the 'Update' method in the code above.

    Anyone had this exception and got a smart solution for it?

    If this is not overcome I think we need to leave EF or any ORM that has this flaw in it.
    Some of the graphs can be very big with references to up to 10 maybe even 20 other entities and those will
    most definitely include same Segement keys.

12. hamid adldoost says:
    December 14, 2014 at 10:20
    There is a problem with this code.. if your entity has a foreign key and the referenced entity exists in database, it will add new copy of that to db because its not attached to the context !

13. Alien says:
    December 19, 2014 at 12:04
    I got Attach working by adding this into 'GenericDataRepository'

    ```
    public void Attach(T entity)
    {
    DbSet dbSet;
    var entry = new MyEntities();
    dbSet = entry.Set();
    ```

dbSet.Attach(entity);
}

Am I correct to assume that this will "only" attach one entity to the context ChangeTracker?
and not a relation to any other entity?

Let's say we got a scenario like this;
Entiy1 = Parent
Entity2 = Child
Entity1 has a one-to-many reference to Entity2 and thus Entity2 many-to-one Entity1, and let's call this downward relation 'children'

What if I want to attach one or more Entity2's to one Entity1, eg, Parent.children.Attach(Tchildren) thus creating the
relation between these?
For me the above code can't be use for that. How can this be done?

14. Adam Hancock says:
May 17, 2015 at 01:12
Excellent article! I've really learn't a lot from this, thank you.

15. sudip purkayastha says:
June 5, 2015 at 01:39
Hi, Can you please share the sample code ?

16. Alien says:
June 5, 2015 at 23:01
@sudip purkayastha
Well he did, there are 2 pages in this thread! and you just posted on page 3.
Check this above: " « Previous 1 2 "

17. Martin says:
September 16, 2015 at 08:44
8. Finally, you need to add a reference from the DAL project to the domain model project in order for it to compile.

I had to do exactly opposite thing, references from Domain to DAL. Not sure if I did something wrong or text is wrong.

18. Magnus Montin says:
September 17, 2015 at 11:43
Hi Martin,

The text is not wrong. The DAL should reference the Mm.DomainModel project where the model/entity classes are defined. The Mm.DomainModel project doesn't reference any other project, it just contains the model/entity classes.

19. Hoss says:
September 21, 2015 at 03:36

Hi Great piece of work and been using it for my projects
I have hit an issue though where you cannot populate records in many-to-many relationships. I have tried the following:
1) When adding parent.childiren.add(c); leads to Primary key violation since tries the related record in the database.
2) Create parent first, reload it and add the children to it. No error but relationship records are not created in the database.

I have seen previous comments regarding sample codes in previous pages but could not find any. Could you please provide the code or guideline on how to fix it.

Thanks

20. nguyen doan says:
September 29, 2015 at 18:08
Hi. i followed your article, but i have a trouble. I used T4 Poco Template to add code generation items to create entities. I want to edit T4 Template to implement interface IEntity but i don't know how to do. Could you please tell me how to do that?

Thanks

21. Juan Rodriguez says:
October 20, 2015 at 21:57
Excellent article Magnus
I actually incorporated your strategy into my project injecting the repository to the controllers with DI and works great.
can you provide your code?

22. Paolo (@___Pool___) says:
November 9, 2015 at 12:36
Consider changing the where predicate from Func in Expression<Func>. If you look at the code in both cases the first get all the entities in context and then filter them in memory (SQL code is generated without the where clause), while using the Expression the sql is genereted with the where clause saving lot of time and effort

23. Jim says:
November 24, 2015 at 18:29
Paolo (@___Pool___) : can you elaborate here? I am running into this problem when trying to use aggregate roots. I am dealing with a big mix of tables, some of which have more than 100 million rows, so in these cases I am extending specific find methods on the repositories, and utilizing sp's when I have big batch updates.

24. Quadrostanology says:
December 14, 2015 at 12:25
Hello I have an issue with the code. The Id is always entry so I got PK violations when I want to add an item to the db.
Please help me out here! :D

25. Mohammed says:
January 6, 2016 at 14:35

Hi Team

Thanks for this nice (Helpful) article
.
I have an addition here

when we Create a new table in database, Model.edmx should be updated from database to add table's entityset.

Model.tt will be regenerate the NewEnitities.cs files depending on it's code generation strategy, in this point we will lose

the modifications which we have made in OldEntities.cs

": IEntity" and "public EntityState EntityState { get; set; }"

therefore, i suggest a solution for this problem

1. Open Model.tt file

2. add these variables at the end of the variables declaration (Lines 13,14)

– var MyAddedCode1 = " : IEntity";
– var MyAddedCode2 = "public EntityState EntityState { get; set; }";

3. find "" and add within the same line.

4. find "" and add after few lines

#>

}

<#

this modification in code generation strategy file will give us the ability to Update OldEntities.cs automatically without losing our modifications which we have made before.

and Thank you again and again for this article

26. Adam Schumph says:
February 1, 2016 at 06:29
Thanks for the great example Magnus; nice to see that something you wrote over two years ago still holds water!

I've been able to use your insights and build out an excellent little nTier solution using EF 6 (database first). I'm wondering if you have any thoughts on report generation? I'm thinking of building a view in the database and using that as a dataset exposed via EF. I would appreciate any thoughts or comments you may have.

Thanks,
Adam

27. Ruben says:
February 18, 2016 at 13:54
Nice example! Just a question how to retrieve properties deeper than one level

In your case you did this:
public Department GetDepartmentByName(string departmentName)
{
return _deptRepository.GetSingle(
d => d.Name.Equals(departmentName),
d => d.Employees); //include related employees
}

What if (for example) your employee has multiple phone numbers stored in another table and you want to include them also?

28. arvindmepani says:
March 18, 2016 at 08:20
How this solution work with unit testing. While developing large application it also reauires to have mock tesing for our data access layer.

29. Murugesan Nataraj says:
April 22, 2016 at 07:54
Good example following commands

public Department GetDepartmentByName(string departmentName)
{
return _deptRepository.GetSingle(
d => d.Name.Equals(departmentName),
d => d.Employees); //include related employees
}
What if (for example) your employee has multiple phone numbers stored in another table and you want to include them also?

30. vit100 says:
April 25, 2016 at 01:30
I believe in method dbSet.Add(item) should be dbSet.Attach(item):

public virtual void Update(params T[] items)
{
using (var context = new Entities())
{
DbSet dbSet = context.Set();
foreach (T item in items)
{
dbSet.Add(item); // <<—-should it be attach here???

foreach (DbEntityEntry entry in context.ChangeTracker.Entries())
{
IEntity entity = entry.Entity;
entry.State = GetEntityState(entity.EntityState);

```
}
}
context.SaveChanges();
}
}
```

31. Sokhawin says:
    May 5, 2016 at 11:00
    Great article, I will apply your guideline in my future development.

32. Bob says:
    May 6, 2016 at 20:45
    Wonderful article, Thank You!

    When adhering to the "I" in S.O.L.I.D, I would like to program against an IEmployee and IDepartment interfaces, but the repositories and Business layer are typed to Employee and Department because the default GenericDataProvider uses the edmx, which needs a concrete type.

    How would I coheres the emdx model into these interfaces so I could use the interfaces everywhere else?

    Thanks again!

    Bob

33. Jim says:
    May 7, 2016 at 20:52
    Bob – correct me if I'm wrong, but since the classes for Employee and Department for example contain only properties, and no methods, I don't think creating interfaces for them gets you much. What would be the methods defined in IEmployee? I think the important part is creating interfaces for the repository classes.

34. Jim says:
    May 7, 2016 at 20:57
    I used Moq to mock the repository classes, and accomplished unit tests in my service layer that way. For the repositories.

35. Bob says:
    May 10, 2016 at 14:36
    Jim,

    I used IEmployee to stick with Magnus' example. My objects (finacial instruments) have many behaviors. So after fetching them with the GenericDataRepository, they will move through many parts of the system.

    It sounds like the GenericDataRepository should handle DTO's only, which would then be used in another concrete class that implements the interfaces (I know there is a pattern for that but can't recal the pattern name!).

    Thanks for your input!

36. Kat Magic says:
    June 27, 2016 at 17:56
    I get an error “interfaces cannot declare types”

37. Kat Magic says:
    June 27, 2016 at 19:52
    I get an error at IQueryable dbQuery = context.Set(); it tells me Set doesn’t exist. Would this be replaced by a newer version of ef? and if so what function should be used instead of Set?

38. Bryce says:
    July 26, 2016 at 19:01
    Any chance we could get the complete sample code uploaded to github or something? I see comments saying its in page 1 or 2 of the comments but still am not having luck finding it.

    Thanks for the article.

39. Cyril says:
    July 29, 2016 at 22:48
    Is there a downloadable codes for the above code?

40. Nirosh says:
    October 8, 2016 at 08:21
    Almost all what you do here is already there in NidoFramework. Why don’t you check that out?

    https://nidoframework.codeplex.com/
    http://www.codeproject.com/Articles/830384/ASP-NET-Csharp-Development-with-Nido-Framework-for

    nuget Install-Package NidoFramework

41. jcfiorenzano says:
    December 8, 2016 at 07:01
    Hi, this is a great article. I have a question in case that you have authentication, where do you put the logic for that, in first place you have to define a user domain that extent from IdentityUser and that would be part of the DomainModel project, so this project would need a dependency with Identity.EntityFramework and does not seem right, another approach could be to put the user domain inside the DataAccessLayer project because this project already have a dependecy with Entity Framework, but it does seem right neither, what do you think that could be a good approach to solve that?