# Signum Framework Tutorials - Part 2: Southwind Logic

## About Signum Framework

Is an application framework for making data-centric windows and web applications. It promotes a code-first workflow and is focused in composability, to share code between projects.

## About this series

In this series of tutorials we will work on a stable application: Southwind.

Southwind is the version of Northwind, the well-known example database provided with Microsoft SQL Server.

In this series of tutorials we will create the whole application, including the entities, business logic, and web (React) user interface, data loading and any other aspect worth to explain.

In this tutorial we will focus on writing the business logic, and how to interact with the database.

## Business Logic

In a Signum Framework application, we understand the business logic as static classes that live in the same assembly (Southwind.csproj) and runs in the server.

This code contains the code that interacts with the database: the business rules, the processes that are defined in our application, the queries that will be used, etc...

For applications with many modules, it is recommended to have one folder for each module, typically with one logic static class and a few highly related entities (like 3 to 10?). This way we keep all the related code together (high cohesion) and we can extract it easily to a different assembly if we want to reuse it.

A Logic static class is responsible for:

* Include these entities in the schema.
* Register associated queries that will be available to the user.
* Define the business logic that will deal with these entities.

## EmployeeLogic: Our first logic class

In opposition of other ORM like NHibernate or Entity Framework, Signum Framework has no concept of session or data context. The change tracking of the entities are kept in the entities itself and other related information, like the current connection or transaction are keep in an implicit contexts (using [***ThreadStatic]/AsyncLocal<T>*** variables).

That allows our logic classes to be **static classes** so they usually have no state and you don’t have to instantiate anything before use them. Also you can easily declare extension methods over the main entity.

All this makes that the business logic looks clean and simple.

Let’s start by writing a class that will deal with employees. We make it a new static class with name EmployeeLogic with the same shape than the example one:

***logic*** [Tab] [Tab] Employee [Enter]

public static class EmployeeLogic

   {

       public static void Start(SchemaBuilder sb)

       {

           if (sb.NotDefined(MethodInfo.GetCurrentMethod()))

           {

               sb.Include<EmployeeEntity>()

                   .WithSave(EmployeeOperation.Save)

                   .WithQuery( () => e => new

                   {

                       Entity = e,

                       e.Id

                       //More fields

                   });

           }

       }

   }

Also, by convention, we make a ‘Start’ static method that take a *SchemaBuilder*, and registers every necessary thing to make Employee a fully working module in our application.

The first thing to do is this method is to check if the method has been already called before, to avoid registering the module twice. We use the convenient *NotDefined* method from *SchemaBuilder*.

In this case, let’s just include EmployeeEntity in our schema. As we see in the last tutorial, just by including EmpoyeeEntity, the related entities (in this case TerritoryEntity and RegionEntity) get automatically included.

### Some business logic

Now let’s create some methods in our logic.

The following line is jut a shortcut to register the Save operation for employee.

.WithSave(EmployeeOperation.Save)

Operations are server side actions registered for an entity that can be called from the user interface, authorized, etc…

If we want to take control of what happens in the server when the user clicks on Save, we can implement the operation manually.

***save*** [Tab] [Tab] Employee [Enter]

Then we can modify the body of the Execute lambda like this:

new Graph<EmployeeEntity>.Execute(EmployeeOperation.Save)

{

CanBeNew = true,

CanBeModified = true,

Execute = (e, \_) =>

{

if (e.IsNew)

{

// When entity is new

}

else

{

// When entity was saved before

};

e.Save();

}

}.Register();

Note that every ***Entity*** has an *IsNew* property that indicates whether the entity has been saved in the database or not.

The Database class contains the methods to interact with the database using Signum’s ORM, like *Save*, *Retrieve* or *Delete* entities. There are generic and untyped overloads of every method, as well as overloads to deal with a single entity or a list of them. Also, thanks to using extensions methods we can keep the code readable.

Database also contains the method *Query<T>()* that opens the door to the Linq provider, the most powerful tool in Signum Framework.

### Linq to Signum

Linq to Signum is robust and complete, you can expect from it pretty much everything that Linq to Sql currently have (*Select, Where, implicit* and *explicit Join, GroupBy, OrderBy, Skip, Take, Single*…) and some new tricks that we will see later.

We can use Linq to Signum in many different situations, typically in your business logic or load application, or you can *expose* open queries to the end user by registering in the QueryLogic (or using WithQuery).

Let’s make a query a little more complex, for example, a method that returns the N best employees that report to the current employee.

public static List<Lite<EmployeeEntity>> TopEmployees(int num)

{

    return (from e in Database.Query<EmployeeEntity>()

            where e.ReportsTo.RefersTo(EmployeeEntity.Current)

            orderby Database.Query<OrderEntity>().Count(o => o.Employee == e.ToLite())

            select e.ToLite()).Take(num).ToList();

}

As you will see, the syntax feels quite natural and similar to other Linq providers with some tiny differences:

* We don’t have an explicit context that represents the database, with a property for each table, instead you will have to use ***Database.Query<T>()*** static method to reach a table of entities. It’s a little longer but has benefits for re-using your code, since you code doesn’t get attached to a particular database schema.
* We use ***Lite<T>*** to represent lazy relationships and, on general, the identity information of an entity. You can generate a Lite<T> from an entity calling ToLite method, and you can compare Entities, Lite<T> arbitrarily using the *Is() method*.

### If you already know other *IQueryable* providers, taking into account these two little differences you should be able to write quite complex queries already.

### How a LINQ provider works?

I would like to get deeper in some issues that affect every Linq-Sql provider and that are not properly explained.

As part of the translation process, your query is basically split in 3 parts.

* Constant sub-expressions that should be evaluated in C# and used as SqlParameters
* A SQL string that represents your query
* A lambda expression that translates each DataRow into your result object.

In our last query, for example, it will get split pretty much like this:

(from e in Database.Query<EmployeeEntity>()

where e.ReportsTo == EmployeeEntity.Current.ToLite()

orderby Database.Query<OrderEntity>().Count(o => o.Employee == e.ToLite())

select e.ToLite()).Take(num).ToList();

It’s important to know this since, if the green or blue code is buggy, then it will look like there’s an exception in the Linq provider, but it’s in your code instead.

Also, writing a Linq provider is quite complex task than needs, apart from implementing the translation of all the Linq operators, to solve some hard problems, some of which should be taken into account by the developer:

* Handle nullability mismatch (explicit in C#, implicit in SQL). Sometimes a cast to *Nullable<T>* is necessary for value types.
* Handle *boolean* expressions (SQL has no *boolean* expressions, just bit values and conditions). These conversions make some SQL queries awkward.
* Short-circuit the evaluation and translation of sub-expression that are known to be useless after the evaluation of a *boolean* constant sub-expression.
* If there are entities in the result set, reconstruct the entity graph without creating duplicates. As an advice, retrieve just *Lite<T>* if possible, its more efficient.
* When the result set has sub-collections on each row, a client join is made automatically (no N+1 problem). Works ok, but take it into account.

### Dynamic queries for the user

The *QueryLogic* contains a repository of queries associated with a key ‘queryName’. These queries are open in the sense that the end user is able to **add filters**, **change order** or even **add and remove columns**.

They are mainly used by the search dialogs in windows and web applications, but the concept is general enough to be used by third party consumers, like the tool for making Charts in Signum Extensions.

Let’s get to work. In the Start method, just after including EmployeeEntity in the Schema, we will register all the entities in the module: EmployeeEntity, RegionEntity and TerritoryEntity.

public static void Start(SchemaBuilder sb)

{

if (sb.NotDefined(MethodBase.GetCurrentMethod()))

{

sb.Include<EmployeeEntity>()

.WithSave(EmployeeOperation.Save)

.WithQuery(() => e => new

{

Entity = e,

e.Id,

e.FirstName,

e.LastName,

e.BirthDate,

e.Photo, //1

});

sb.Include<RegionEntity>()

.WithSave(RegionOperation.Save)

.WithQuery(() => r => new

{

Entity = r,

r.Id,

r.Description,

});

sb.Include<TerritoryEntity>()

.WithSave(TerritoryOperation.Save)

.WithQuery(() => t => new

{

Entity = t,

t.Id,

t.Description,

t.Region

});

}

}

Let’s analyze. For each type we are:

* Include: Including the table in the database Schema.
* WithSave: Register the Save operation (in OperationLogic)
* WithQuery: Register the Query (in QueryLogic).

WithSave and WithQuery are just shortcuts for registering simple operations and queries respectively. But we could take register the Employee custom query like this.

QueryLogic.Queries.Register(typeof(EmployeeEntity), () =>

from e in Database.Query<EmployeeEntity>()

from t in e.Territories

select new

{

Entity = e,

e.Id,

e.FirstName,

e.LastName,

e.BirthDate,

e.Photo, //1

});

Here we can see that what we do is to associate each query with a type, in the first case *typeof(EmployeeEntity)*.

Since a queryName is an object, we can use a *System.Type* as a name. When a query is associated with a type it becomes the *default query* for this type.

In order to create a dynamic query, the only necessary thing is to provide an *IQueryable<T>* that contains an Entity property of type *Lite<T>*. This value will be the entity associated with every record and won’t be shown to the end user.

By convention, other properties are exposed as the default columns of the query, but the user is free to remove them and add other columns.

The new columns could be any field of the entity and, if the field is another entity, the fields of the sub entity and so on. A left outer join will be made, so the number of rows won’t change.

In the case of collection fields, however, we cannot show the related sub-entities without multiplying the number of rows, so a different query is necessary:

QueryLogic.Queries.Register(EmployeeQuery.EmployeesByTerritory, () =>

from e in Database.Query<EmployeeEntity>()

from t in e.Territories

select new

{

Entity = e,

e.Id,

e.FirstName,

e.LastName,

e.BirthDate,

e.Photo, //2

Territory = t,

});

Note how in this new query we used an enum -defined after the entity- as the key of the query. We recommend enums instead of plain strings because they are type safe and easier to localize.

On the right side, note that we used a double from, this is translated as a *SelectMany* by the compiler and will multiply the number of rows.

## ProductLogic: expression properties

Let’s make now a *ProductLogic* static class with a Start method starting with If *NotDefined*.

Then we include ProductEntity in the schema builder, and register the operation and the query:

sb.Include<ProductEntity>()

.WithSave(ProductOperation.Save)

.WithQuery(() => p => new

{

Entity = p,

p.Id,

p.ProductName,

p.Supplier,

p.Category,

p.QuantityPerUnit,

p.UnitPrice,

p.UnitsInStock,

p.Discontinued

});

And the sane for Categories:

sb.Include<CategoryEntity>()

.WithSave(CategoryOperation.Save)

.WithQuery(() => s => new

{

Entity = s,

s.Id,

s.CategoryName,

s.Description,

s.Picture

});

And Suppliers:

sb.Include<SupplierEntity>()

.WithSave(SupplierOperation.Save)

.WithQuery(() => s => new

{

Entity = s,

s.Id,

s.CompanyName,

s.ContactName,

s.Phone,

s.Fax,

s.HomePage,

s.Address

});

And let’s make two queries for products so we don’t have to filter-out discontinued products all the time:

                QueryLogic.Queries.Register (ProductQuery.CurrentProducts, () =>

                    from p in Database.Query<ProductEntity>()

                    where !p.Discontinued

                    select new

                    {

                        Entity = p,

                        p.Id,

                        p.ProductName,

                        p.Supplier,

                        p.Category,

                        p.QuantityPerUnit,

                        p.UnitPrice,

                        p.UnitsInStock,

                    });

We won’t create any method for dealing with the entities, the default behavior of the user interface will just Save the objects and this is all right in this case.

Quite simple, now let’s complicate it a little bit more:

Let’s suppose for the administrator would be useful to have the *ValueInStock* column *(UnitPrice \* UnitsInStock)* to avoid accumulating too much capital in products.

We could just create the column in the query, but then we would have to replicate the code to show the value in the user interface of the entity, in the business logic or in some reports.

A much more elegant solution would be to add a read–only calculated property in the product entity itself, like this:

public class ProductEntity : Entity

{

(…)

public decimal ValueInStock

{

get { return UnitPrice \* UnitsInStock; }

}

(…)

}

Unfortunately, the body of the getter of this property will be a normal method compiled in IL, so the LINQ provider won’t have an idea of what *ValueInStock* means.

Using more modern C# we could use an expression-bodied propertly like this:

public class ProductEntity : Entity

{

(…)

public decimal ValueInStock => UnitPrice \* UnitsInStock;

}

(…)

}

But this is just syntax sugar by the C# compiler. The same method will be compiled to IL.

What we would like to do is to keep the body as an *Expression Tree (Expression<T>)*, so the LINQ provider can understand the definition.

Since C# has no way to declare methods implemented ad expression trees, Signum did a lot of black magic to implement it.

Just write:

***expressionProperty*** [Tab] [Tab] ProductEntity [Tab] decimal [Tab] ValueInStock [Tab] p [Enter]

and finally, we will implement the body of the lambda with our simple formula:

[AutoExpressionField, Unit("$")]

public decimal ValueInStock => As.Expression(() => unitPrice \* unitsInStock);

Voilà, with the help of AutoExpressionFiel attribute and As.Expression method we can declare a method that will be understood by the LINQ provider and also works in-memory.

**How the Black Magic works?**

Under the covers AutoExpressionField and As.Expression is recognized by the NuGet Signum.MSBuildTask and replaced at compile-time into something like:

public class ProductEntity: Entity

{

(…)

static Expression<Func<ProductEntity, decimal>> ValueInStockExpression =

p => p.UnitPrice \* p.UnitsInStock;

[ExpressionField(nameof(ValueInStockExpression))]

public decimal ValueInStock

{

get { return ValueInStockExpression.Evaluate(this); }

}

(…)

}

As you see, the body of the expression is now moved to a static field of type Expression<T> with similar arguments / return type, including the object itself because ValueInStock is an instance property.

Also a new ExpressionFieldAttribute in the property points to the new static field.

This attribute will be used at run-time by the LINQ provider to find the implementation of a method when translating a query to SQL.

The implementation of the ValueInStock property also calls the new ValueInStockExpression field using the Evaluate method. This method (defined in *Signum.Utilities*) compiles, catches and invokes the lambda expression, so we only have to write the code once.

Knowing how this black magic works is useful if you need the definition for the queries to be different than the one for code (for example using a cache dictionary): Just implement the pattern with ExpressionFieldAttribute and write a different code in the getter.

By using this, you can add more semantic information to your entities and factor-out your business logic in reusable functions that can be understood by the queries, making your business logic way simpler and easier to maintain.

This technique works for properties and methods (even extension methods) and for static and instance members, and constitutes the simplest way to expand the LINQ provider (there are two ways more).

### How the expression trees get generated?

The Linq syntax for querying in-memory objects is so similar to the one for querying the database that sometimes developers get confused. The reason is that creating an expression tree looks exactly as creating a normal lambda expression.

An expression tree is a run-time representation of a piece of code that can be used by a library (like a LINQ provider). This expression trees can be compiled an executed, but compiled code cannot be converted to expression trees. Only expression trees can be translated to SQL.

The code that creates this tree of nodes is plain C# that uses the Expression class, and can be automatically generated by the C# compiler, or can be written manually.

Currently the C# compiler is able to generate expression trees for lambda expressions witch inferred type is *Expression<T>*, being *T* some delegate type, and only If the lambda expression has expression body (instead of statement body).

However, when we call the methods on *Queryable* static class (*Select, Where, GroupBy*…), the methods are actually executed (contrary to common belief) but their only purpose is to create another *IQueryable<T>* with an expression that ‘extends’ the expression of the previous *IQueryable<T>* with the node for the current operator (code that writes itself!).

For example, in our previous query example (once without query comprehensions):

Database.Query<EmployeeEntity>()

.Where(e=> e.ReportsTo == EmployeeEntity.Current.ToLite())

.OrderBy(e=> Database.Query<OrderEntity>().Count(o => o.Employee == e.ToLite()))

.Select(e=> e.ToLite())

.Take(num)

.ToList();

* Expression created by the C# compiler
* Expression created at runtime

This subtle difference is important in the following situations: Imagine that we make an extension method like this:

public static IQueryable<ProductEntity> AvailableOnly(this IQueryable<ProductEntity> products)

{

return products.Where(p => !p.Discontinued);

}

This code will work all right as long as we use it in the ‘main query path’ because it will get executed, but if we use it inside of a lambda expression (i.e: where predicate), then the code is not executed and the LINQ provider finds an *AvailableOnly* method but has no clue what it does.

Using expressionMethods we circumvent this problem:

static Expression<Func<IQueryable<ProductEntity>, IQueryable<ProductEntity>> AvailableOnlyExpression = products => products.Where(p => !p.Discontinued);

public static IQueryable<ProductEntity> AvailableOnly(this IQueryable<ProductEntity> e)

{

return AvailableOnlyExpression.Evaluate(this);

}

Note: In more complex scenarios, like when the method is generic or there are different overloads, we could use *MethodExpanderAttribute* and *IMethodExpander* to teach the LINQ provider how to translate the unknown method.

## CustomerLogic: inheritance and ImplementedBy

Every ORM has to deal with inheritances somehow (Table per hierarchy, Table per subclass, Table per concrete class).

Signum Framework keeps every concrete class in his own individual table, and uses polymorphic foreign keys for modeling inheritance, two different attributes we can place on any field that represent a relationship (*Entity* or *Lite<T>*) in our entities:

* **ImplementedBy**: creates a set of mutually exclusive foreign keys to differentiate tables that correspond to just one field of an entity. Useful when there are just a few different implementations.
* **ImplementedByAll**: creates two columns (id and typeId), allowing to point to any entity in the database, but has no referential integrity and a weaker support from the UI and the LINQ provider (no automatic union).

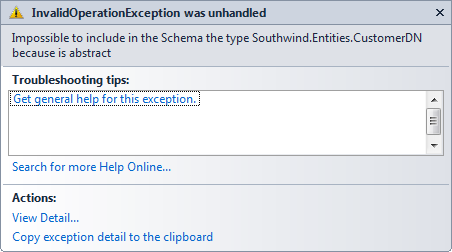
This solution has some important advantages over other solutions:

* Every entity has just one *Type* and *Id*, and lives in just one table. In other words, he Cat with Id 4 has no Id as an Animal.
* The database doesn’t need to know about all the hierarchy of classes (like abstract classes) only the concrete ones that are included in the schema.
* Since we have a way to override attributes -even without control of the entity- we can use polymorphic foreign keys to add expansion points in our modules.

Northwind was a traditional schema without any concept of inheritance or polymorphism. In our model, what we will do is to pretend than now Southwind has companies and persons as customers, with some common data and some different one, let’s go back to our entities and make some changes:

1. Make CustomerEntity abstract and create two new entities: PersonEntity and CompanyEntity, both inheriting from CustomerEntity.
2. Move CompanyName, ContactName and ContactTile to CompanyEntity.
3. Add some properties to PersonEntity: Title, FirstName, LastName, and DateOfBirth (with DateTimePrecissionValidator to Days).
4. Override *ToString* in both classes returning the CompanyName for CompanyEntity, and the concatenation of first and last name for PersonEntity.

If we try to start Southwind.Terminal application in this moment, we will get an Exception like this:



As we see, we cannot include abstract classes (or interfaces) in the Schema, only concrete classes that can be instantiated.

In order to fix the error we need to indicate to the *SchemaBuilder* that whenever he found a relationship to a CustomerEntity, there will be either a PersonEntity or a CompanyEntity. If we have control of the class the easiest way is to add an *ImplementedByAttribute* in the **Customer** **field** in OrderEntity:

[NotNullable]

[NotNullValidator]

[ImplementedBy(typeof(CompanyEntity), typeof(PersonEntity))]

public CustomerEntity Customer { get; set; }

### Syncronizing

Let’s try again and now we should be able to synchronize the schema.

First the synchronizer detects that CustomerEntity table has been removed and that some new tables (CompanyEntity and PersonEntity) have been created, and ask if CustomerEntity has been renamed. We currently have no data inside so it doesn’t really matter, but if we would have some, probably we would like to move it to CompanyEntity, so let’s pretend and answer CompanyEntity.

Secondly, he asks for the CutomerID field in OrderEntity, now we would have *CustomerID\_Company* and *CustomerID\_Person*, if we would like to keep the data we would have to choose the first one.

Wow! Here is our sync script automatically generated!

DROP INDEX Order.FIX\_Order\_CustomerID;

ALTER TABLE Order DROP CONSTRAINT FK\_Order\_CustomerID ;

EXEC SP\_RENAME 'Customer' , 'Company';

ALTER TABLE Company ALTER COLUMN ContactTitle NVARCHAR(10) NOT NULL;

EXEC SP\_RENAME 'Order.CustomerID' , 'Customer\_Company', 'COLUMN' ;

ALTER TABLE Order ADD CustomerID\_Person INT NULL -- DEFAULT( );

CREATE TABLE Person(

Id INT IDENTITY NOT NULL PRIMARY KEY,

ToStr NVARCHAR(200) NULL,

Ticks BIGINT NOT NULL,

Address\_HasValue BIT NOT NULL,

Address\_Address NVARCHAR(60) NULL,

Address\_City NVARCHAR(15) NULL,

Address\_Region NVARCHAR(15) NULL,

Address\_PostalCode NVARCHAR(10) NULL,

Address\_Country NVARCHAR(15) NULL,

Phone NVARCHAR(24) NOT NULL,

Fax NVARCHAR(24) NOT NULL,

FirstName NVARCHAR(40) NOT NULL,

LastName NVARCHAR(40) NOT NULL,

Title NVARCHAR(10) NOT NULL,

DateOfBirth DATETIME NOT NULL

);

ALTER TABLE Order ADD CONSTRAINT FK\_Order\_CustomerID\_Company FOREIGN KEY (CustomerID\_Company) REFERENCES Company(Id);

ALTER TABLE Order ADD CONSTRAINT FK\_Order\_CustomerID\_Person FOREIGN KEY (CustomerID\_Person) REFERENCES Person(Id);

CREATE INDEX FIX\_Order\_CustomerID\_Company ON Order(CustomerID\_Company);

CREATE INDEX FIX\_Order\_CustomerID\_Person ON Order(CustomerID\_Person);

UPDATE Type SET --Customer

ToStr = 'Company',

FullClassName = 'Southwind.Entities.Company',

TableName = 'Company',

CleanName = 'Company',

FriendlyName = 'Company'

WHERE id = 3;

INSERT TypeDN (ToStr, FullClassName, TableName, CleanName, FriendlyName)

VALUES ('Person', 'Southwind.Entities.Person', 'Person', 'Person', 'Person');

As you can see, the script removes and creates indexes, foreign keys, tables, and add the necessary records so Signum Framework can be happy with the database.

This feature makes working in group really nice, since each member doesn’t have to maintain a script with the necessary changes and merge it with the rest, a very error prone process. Just create the Sync script, check no data will get lost and run it.

In the case when a not null field is added to a table, a commented DEFAULT constraint is written in the sentence for you to define the default value. If the default value depends on some other data you can temporally make the field *nullable*, fill the data, and then generate the synchronization script again.

In this case we have no records so we can just remove the DEFAULT constraints and run the script.

Ok, let’s create our CustomerLogic (actually we haven’t even created it yet, all this was going on because we included OrderEntity in MyEntityLogic).

As usual, a static class with a Start method, we include CompanyEntity and PersonEntity and add some default queries for them. The result should be like this:

public static void Start(SchemaBuilder sb, DynamicQueryManager dqm)

    {

            if (sb.NotDefined(MethodInfo.GetCurrentMethod()))

            {

                sb.Include<PersonEntity>();

                sb.Include<CompanyEntity>();

                dqm.RegisterQuery(typeof(PersonEntity), () =>

                    from r in Database.Query<PersonEntity>()

                    select new

                    {

                        Entity = r,

                        r.Id,

                        r.FirstName,

                        r.LastName,

                        r.DateOfBirth,

                        r.Phone,

                        r.Fax,

                        r.Address,

                    });

                dqm.RegisterQuery(typeof(CompanyEntity), () =>

                    from r in Database.Query<CompanyEntity>()

                    select new

                    {

                        Entity = r,

                        r.Id,

                        r.CompanyName,

                        r.ContactName,

                        r.ContactTitle,

                        r.Phone,

                        r.Fax,

                        r.Address,

                    });

}

}

Finally, in the case that we wouldn’t have control of the OrderEntity entity or CustomerEntity entity (because they are implemented by a third party) we could still override the attributes at the beginning of our Starter (the global one) like this:

sb.Settings.OverrideFieldAttributes((OrderEntity o) => o.Customer,

new ImplementedByAttribute(typeof(CompanyEntity), typeof(PersonEntity)));

## OrderLogic: the last step

Our last step is to create the OrderLogic to match its own current responsibility and add a default query for it.

public static void Start(SchemaBuilder sb, DynamicQueryManager dqm)

       {

           if (sb.NotDefined(MethodInfo.GetCurrentMethod()))

           {

               sb.Include<OrderEntity>();

               dqm.RegisterQuery(typeof(OrderEntity), () =>

                   from o in Database.Query<OrderEntity>()

                   select new

                   {

                       Entity = o,

                       o.Id,

                       o.State,

                       o.Customer,

                       o.Employee,

                       o.OrderDate,

                       o.RequiredDate,

                       o.ShipAddress,

                       o.ShipVia,

                   });

               dqm.RegisterQuery(OrderQuery.OrderSimple, () =>

                   from o in Database.Query<OrderEntity>()

                   select new

                   {

                       Entity = o,

                       o.Id,

                       o.State,

                       o.Customer,

                       o.Employee,

                       o.OrderDate,

                       o.RequiredDate,

                       o.ShipAddress,

                       o.ShipVia,

                   });

}

}

We also need to create a query that shows the content of every order (OrderLines). Again, we will use an enum for the key.

dqm.RegisterQuery(OrderQuery.OrderLines, () =>

                   from o in Database.Query<OrderEntity>()

                   from od in o.Details

                   select new

                   {

                       Entity = o,

                       o.Id,

                       od.Product,

                       od.Quantity,

                       od.UnitPrice,

                       od.Discount,

                       od.SubTotalPrice,

                   });

These queries are all right, but wouldn’t be useful to compute the *SubTotalPrice* for each OrderDetail, as well as the TotalPrice for an order.

We can do it using expressionProperty again, so we have the property available in any scenario.

[Serializable]

public class OrderDetailEmbedded : EmbeddedEntity

{

(…)

[AutoExpressionField]

public decimal SubTotalPrice => As.Expression(() => Quantity \* UnitPrice \* (1 - Discount));

(…)

}

Finally, let’s write a little bit of business logic that actually does something.

Let’s suppose every time a new OrderEntity is created in the database, the products have to be removed from the stock. If the user interface tries to create an order with more products than the available ones, it should throw an exception to abort the operation.

Frist we should add a new operation to the OrderEntity as following:

[AutoInit]

    public static class OrderOperation

    {

        public static readonly ExecuteSymbol<OrderEntity> Save;

(…)

    }

Then in the logic file we should write the following codes:

new Graph<OrderEntity>.Execute(OrderOperation.Save)

    {

CanBeNew = true,

          CanBeModified = true,

          Execute = (o, \_) =>

{

if (o.IsNew)

{

foreach (var od in o.Details)

{

int updated = od.Product.InDB()

.Where(p => p.UnitsInStock >= od.Quantity)

.UnsafeUpdate(p => p.UnitsInStock,   
 p => (short)(p.UnitsInStock - od.Quantity);

if (updated != 1)

throw new ApplicationException("There are not enough {0} in stock".FormatWith(od.Product));

}

}

o.Save();

}

    }.Register();

Inside of the *foreach* loop we can see a new ***UnsafeUpdate*** method. *UnsafeUpdate* (and *UnsafeDelete*) is a lightweight way of modifying and removing records in the database. It mimics SQL UPDATE and DELETE syntax so it’s fast, but knows about all the conventions and primitives of the framework (*EmbeddedEntities, enums, Lites, ImplementedBy…*) on the other side it doesn’t test any validation, that’s why it’s called *Unsafe.*

*UnsafeDelete* takes and *IQueryable<T>* where *T* is a concrete entity type, so it can be used after any query. In this case we are using another useful new friend: *InDB*.

This method creates an *IQueryable* from a in-memory entity or Lite, and its equivalent to:

od.Product.InDB() 🡪 Database.Query<ProductEntity>().Where(p => p == od.Product)

Finally, *UnsafeUpdate* returns the number of rows modified. In this case *InDB* already select just one product, so if the Where filters the product and we don’t update any row, we know that we don’t have enough stock and we can throw an exception.

## Putting everything together

The last step is to add all this modules to your application.

We have seen that every Logic module has a Start method, responsible of adding the necessary tables to the schema, registering queries, hooking events and so one…

There’s a global *Start* method, in the *Starter* class, this method is responsible for starting all the modules you are going to use in your application and call to their Start method.

Is a good practice to register these methods in order of dependency -from the less dependent to the more dependant- so let’s start our modules:

public static class Starter

{

public static void Start(string connectionString)

{

(…)

SchemaBuilder sb = new SchemaBuilder();

DynamicQueryManager dqm = new DynamicQueryManager();

sb.Schema.ForceCultureInfo = CultureInfo.InvariantCulture;

ConnectionScope.Default = new Connection(connectionString, sb.Schema, dqm);

(…)

EmployeeLogic.Start(sb);

    ProductLogic.Start(sb);

   CustomerLogic.Start(sb);

    OrderLogic.Start(sb);

    ShipperLogic.Start(sb);

(…)

}

}

After the *Start* Method gets called your application scope is defined and we are ready to Generate the Database, *Synchronize* or try to *Initialize* the application.

## Conclusion

In this long article we learn a lot about how to write the business logic using Signum Framework, for example:

* Create our own business logic in a modular way
* How to Save, and Retrieve objects
* Use Linq to Signum to query the database
* Create our own database-enabled calculated properties using *expressionProperty* template
* Using *ImplementedBy* to represent polymorphic relationships of a hierarchy of classes.
* *Synchronize* the database
* Using Transactions
* Use *UnsafeUpdate* to change values in the database without any validation ceremony
* How the Start/Initialization sequence works

Of course there are things that are left to explain about how to write the Logic classes:

* Getting the control before and entity is Saved or Retrieved using EntityEvents or overriding methods in the entity itself.
* Hooking into the synchronization and generation of the database to include your own specific scripts.

And, once *Signum.Extensions* gets published, we could see how the Authorization module is able to protect any resource in the database (types, properties, queries…) and how Operation, Processes and ScheduledTask modules simplify writing the business logic.

In the next tutorial we will use Southwind.Terminal to move the data from Northwind database using Linq to Sql and CSV files.

Hopefully, soon we will see window with a button that can actually *do something*.