# Signum Framework Course - Part 1: Southwind Entities

## About this series

In order to show the capabilities of the framework, and have a good understanding of the architecture, we are preparing a series of tutorials in which we will work on a stable application: Southwind.

Southwind is the Signum 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, web (React) user interface, data loading and any other aspect worth to explain.

Time to get our hands dirty with the entities:

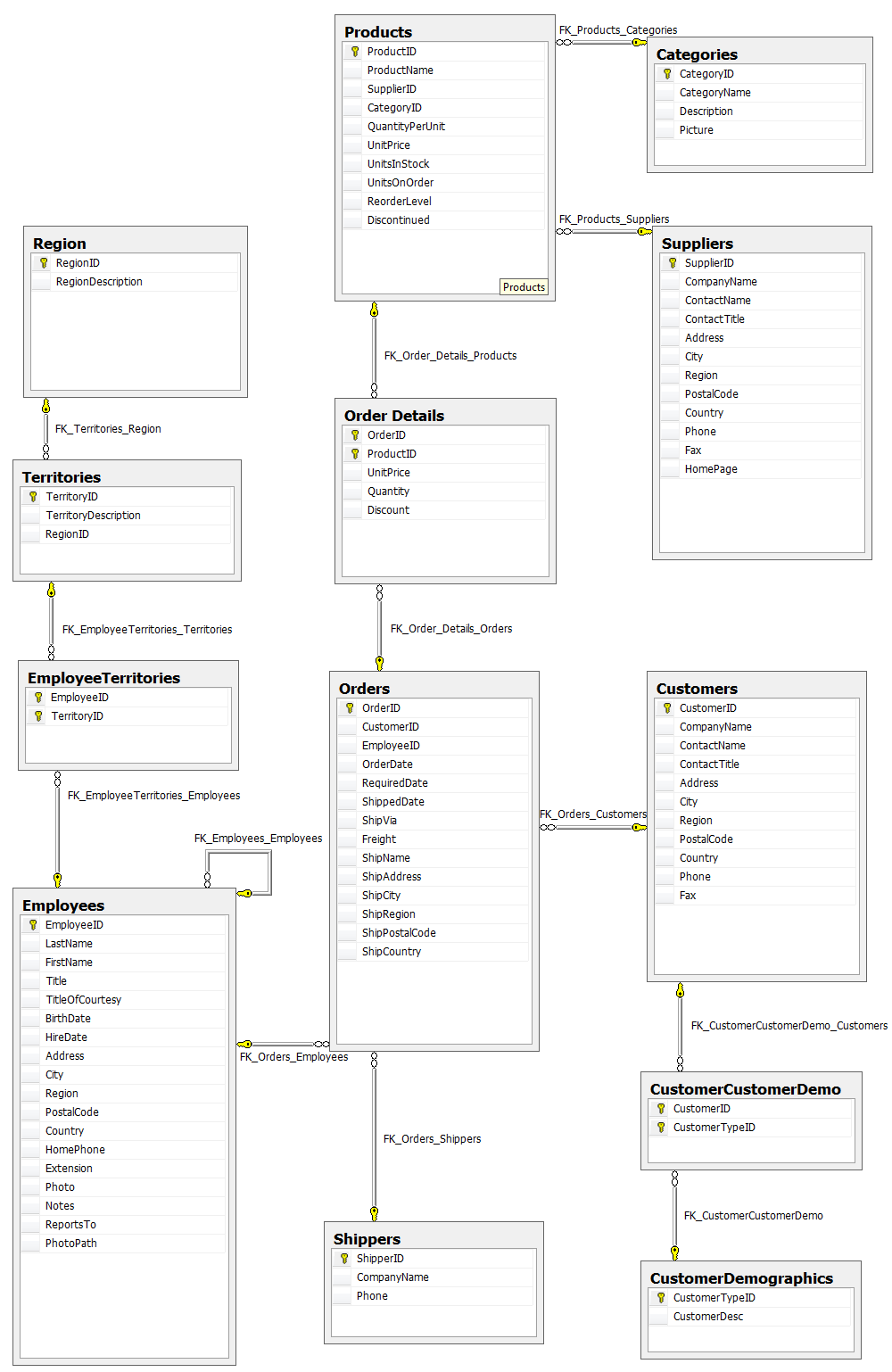
In Visual Studio 2017 project we have the following projects:

* Southwind.Entities
* Southwind.Load
* Southwind.Logic
* Southwind.React
* Southwind.Windows

## Creating the entities

Signum Framework promotes a code-first approach and the entities you write have a straightforward mapping to database tables. Also, since 100% of the SQL queries are produced by the framework, including schema modification, you almost forget about SQL Management Studio.

For teaching, however, will be easier to start by showing what we are trying to accomplish in a familiar diagram. Here is Northwind database:



## Out first entity: Region

Let’s start simple. In order to create the entity for Region we need to inherit from Entity class and create the description field.

We will call it Description, not RegionDescription, since this redundancy makes sense only to simplify writing SQL manually (not the case).

Also, we don’t have to worry about RegionID, since **every Entity already has Id and ToStr field/property.**

We already have the snippets installed so just remove Region class and press:

***entityWithName*** [Tab] [Tab] Region [Tab] Description [Enter]

After this, we should have something like this:

[Serializable, EntityKind(EntityKind.String, EntityData.Master)]

    public class RegionEntity : Entity

    {

        [StringLengthValidator(Min = 3, Max = 100), UniqueIndex]

        public string Description { get; set; }

        [AutoExpressionField]

        public override string ToString() => As.Expression(() => Description);

    }

    [AutoInit]

    public static class RegionOperation

    {

        public static readonly ExecuteSymbol<RegionEntity> Save;

    }

This is our entity class, some things to note:

### The class

The class is Serializable, so you can send it though a web service or save it in a file.

Also, inherits from Entity to enables concurrency support.

It is mandatory to suffix the entity’s name with “Entity”. It is a name convention that will considered by the framework in both *Client* and *Server* parts.

Finally, note that we write the name of our entities in singular, as well as the name of the table.

### The field

There’s a description field. In Signum Framework entities, (almost) every field will generate a database column.

The column will try to match the CLR type whenever possible so in the case of strings it will be nullable and, by default, a length of 100 characters.

The attributes over the field override some of this defaults, in this case makes the column not nullable and with a length of 200.

Just by placing *UniqueIndex* attribute, an index will be created over the column.

The *UniqueIndex* attribute could has *AllowMultipleNulls* as input parameter when your filed is nullable.

### The property

Every property gives access to the underlying field for the user interface and business logic.

By decorating the properties with ValidationAttributes we enforce simple validation rules over the entities. More flexible validation options are available and the attributes can be overridden.

Also we can decorate our properties with other annotations to change the display name, the format of numbers or dates, the unit of the value, etc…

Entity with foreign key: Territory

Let’s continue now with Territory:

***entityWithName*** [Tab] [Tab] Territory [Tab] Description [Enter]

We also change the base class to Entity, TerritoryID comes for free but we have to create the RegionID column and the foreign key. Quite simple, just create a property of type Region:

***fieldLite*** [Tab] [Tab] RegionEntity [Tab] Region [Enter]

Since Region property is mandatory, let’s add NotNullable and NotNullValidator over the property. The result should be something like this:

[Serializable, EntityKind(EntityKind.String, EntityData.Master)]

    public class TerritoryEntity : Entity

    {

        [StringLengthValidator(Min = 3, Max = 100), UniqueIndex]

        public string Description { get; set; }

        public Lite<RegionEntity> Region { get; set; }

        [AutoExpressionField]

        public override string ToString() => As.Expression(() => Description);

    }

    [AutoInit]

    public static class TerritoryOperation

    {

        public static readonly ExecuteSymbol<TerritoryEntity> Save;

    }

The next step should be the EmployeeTerritories relational table, this table however is not an ‘Entity’ but a Many-to-Many relationship between Employees and Territories. In Signum Framework this is represented by a MList<TerritoryEntity> field on Employee entity.

EmbeddedEntity: Address

Before getting into Employee table, we can see that some fields (Address, City, Region, PostalCode and Country) are also repeated in Customers, Orders and Supplier tables.

In order to create an address entity that ‘belongs’ to the parent table we have to make it inherit from EmbeddedEntity:

***embeddedEntity*** [Tab] [Tab] Address [Enter]

***fieldString*** [Tab] [Tab] [Tab] Address [Enter]

Do the similar for City, Region, PostalCode and Country properties. Now let’s make some changes to the size of the fields (and the corresponding validators) to mimic the Northwind database.

[Serializable]

    public class AddressEmbedded : EmbeddedEntity

    {

        [StringLengthValidator(Min = 3, Max = 60)]

        public string Address { get; set; }

        [StringLengthValidator(Min = 3, Max = 15)]

        public string City { get; set; }

        [StringLengthValidator(Min = 3, Max = 15)]

        public string Region { get; set; }

        [StringLengthValidator(Min = 3, Max = 10)]

        public string PostalCode { get; set; }

        [StringLengthValidator(Min = 3, Max = 15)]

        public string Country { get; set; }

    }

Note: It is mandatory to end with “Embedded” while naming. It is a name convention that will considered by the framework in both *Client* and *Server* parts.Getting big: Employee entity

Now things get interesting, Employee it’s one of the biggest tables on Northwind and has some new aspects:

* An Address field, of type AddressEmbedded, that contains the fields of our new EmbeddedEntity
* An MList<TerritoryEntity> that will create the relational table.
* A relationship to itself to represent the Employee hierarchy.
* A bunch of new value types like DateTime, DateTime? (nullable), byte[] for the photo, and a infinite length string (notes).

We should have the basis of the entity written in a few seconds with the following key strokes:

***entity*** [Tab] [Tab] EmployeeEntity [Enter]

***fieldString*** [Tab] [Tab] [Tab] LastName [Enter]

***fieldString*** [Tab] [Tab] [Tab] FirstNaame [Enter]

***fieldString*** [Tab] [Tab] [Tab] Title?[Enter]

***fieldString*** [Tab] [Tab] [Tab] TitleOfCourtesy? [Enter]

***field*** [Tab] [Tab] DateTime? [Tab] BirthDate [Enter]

***field*** [Tab] [Tab] DateTime? [Tab] HireDate [Enter]

***field*** [Tab] [Tab] AddressEmbedded [Tab] Address [Enter]

***fieldString*** [Tab] [Tab] [Tab] HomePhone? [Enter]

***fieldString*** [Tab] [Tab] [Tab] Extension? [Enter]

***field*** [Tab] [Tab] byte[] [Tab] Photo? [Enter]

***fieldString*** [Tab] [Tab] [Tab] Notes? [Enter]

***fieldLite*** [Tab] [Tab] EmployeeEntity [Tab] ReportsTo? [Enter]

***fieldString*** [Tab] [Tab] [Tab] PhotoPath? [Enter]

***fieldMList*** [Tab] [Tab] TerritoryEntity [Tab] Territories [Enter]

### The class

This time inheriting from Entity is all right. So we have something like this:

[Serializable, EntityKind(EntityKind.Main, EntityData.Master)]

public class EmployeeEntity : Entity

{

    [StringLengthValidator(Min = 3, Max = 20)]

    public string LastName { get; set; }

    [NotNullable, SqlDbType(Size = 10)]

    [StringLengthValidator(Min = 3, Max = 10)]

    public string FirstName { get; set; }

    [StringLengthValidator(Min = 3, Max = 30)]

    public string? Title { get; set; }

    [StringLengthValidator(Min = 3, Max = 25)]

    public string? TitleOfCourtesy { get; set; }

    [DateTimePrecissionValidator(DateTimePrecision.Days)]

    public DateTime? BirthDate { get; set; }

    public DateTime? HireDate { get; set; }

    public AddressEmbedded Address { get; set; }

    [StringLengthValidator(Min = 3, Max = 25), TelephoneValidator]

    public string? HomePhone { get; set; }

    [StringLengthValidator(Min = 3, Max = 4), TelephoneValidator]

    public string? Extension { get; set; }

    public Lite<FileEntity>? Photo { get; set; }

    [StringLengthValidator(MultiLine = true)]

    public string? Notes { get; set; }

    public Lite<EmployeeEntity>? ReportsTo { get; set; }

    [StringLengthValidator(Min = 3, Max = 255), URLValidator]

    public string? PhotoPath { get; set; }

    [NoRepeatValidator]

    public MList<TerritoryEntity> Territories { get; set; } = new MList<TerritoryEntity>();

    public override string ToString()

    {

        return "{0} {1}".FormatWith(FirstName, LastName);

    }

}

[AutoInit]

public static class EmployeeOperation

{

    public static ExecuteSymbol<EmployeeEntity> Save;

}

Let’s manually set the size of all the string fields.

### DateTime fields

Not that for DateTime fields we just make the field type nullable when is not mandatory.

Also, we could place a convenient [DateTimePrecissionValidator(DateTimePrecision.Days)] that simplifies constraining our dates avoiding rounding errors. Note that all the ValidatorAttributes, by convention, let null values go through and in order to prevent null values you will need a [NotNullValidator] as well. NotNullValidator is added automatically for any reference type that is not-nullable.

### NVarChar(MAX) fields

We could override the field ‘Notes’ to have an NText type with the attribute

SqlDbType(SqlDbType = SqlDbType.NText)

But since NText is already deprecated in favor of NVarChar(MAX), we will place this attribute instead:

SqlDbType(Size = int.MaxValue)

Other string fields, like HomePhone and extension, give us the opportunity to use one

### VarBinary(MAX) fields

The same is applicable to ‘Photo’ field. By default byte[] fields are translated to VarBinary. Instead of using SqlDbType.Image, we will use Size = int.MaxValue to make it VarBinary(MAX). Most of the times we use Lite<FileEntity>.

### EmbeddedEntity fields

The address field, of type AddressEmbedded, will include all the corresponding columns in the form Address\_Address, Address\_City, Address\_Region, ....

Also in order to express that the address entity itself is null (not some of the internal fields), a new Address\_HasValue field will be created and the types of the internal fields will be overridden to support null values.

We can disable this feature just by placing [NotNullable] over the ‘address’ field, but in this case this is all right.

### Lite fields

*Lite<T>* is a generic class that creates a lightweight version of an entity. It only contains a Type, Id and ToStr fields.

*Lite<T>* can be used in your entity model to control lazy retrieving of entities and it will have no difference in the schema.

Also *Lite<T>* can be used in your business logic to pass it as a parameter, in your queries, or in the user interface, to populate a combo box for example. We will see *Lite<T>* many times in the course of this tutorial.

In this case, by making ‘ReportsTo’ a field of type *Lite<EmployeeEntity>* we stop the engine from retrieving every employee all the way up in the chain of command.

### MList<TerritoryEntity> fields

Finally, the field ‘Territories’ of type *MList<TerritoryEntity>* won’t create any column. Instead it will create a Table with name EmployeeTerritories that will look quite similar to the original one.

Note that MLists are not only used to create relational tables (collection of other entities) but can also be used to create collection of values or collections of embedded entities.

Getting fast: Rest of the entities

After writing Employee entity, writing the rest of the entities should be straightforward.

Maybe it gets a little boring, but this is only for teaching purposes. In a real application you will create the entities **instead** of the tables, not **after**.

LEGACY APPS  
For many years we resist making a tool that generates the entities from a legacy database automatically. Signum Framework was quite strict with some conventions that made it hard to consider any random legacy database. Also, writing the entities by hand is a good opportunity to reconsider de design and fix legacy mistakes.

Around summer 2014, however, we relaxed some of these conventions and create the tooling to make it possible to create an application on top of a legacy app. If you are interested check the Adventure Works tutorial in [www.signumframework.com](http://www.signumframework.com).

Some small notes:

* **Shipper** (straightforward)
* **Customer**
  + Use our AddressEmbedded embedded entity.
  + Skip Customers demographics since the table is empty and adds no value (it would be just an MList of CustomerDemographicsEntity)
* **Supplier**
  + Use our AddressEmbedded
  + Use URLValidators on HomePage
  + Use TelephoneValidator on Phone and Fax
* **Product**
  + Make the relationship to CategoryEntity and SupplierEntity, both a Lite<T> relationship.
* **Category**
  + Use a *Lite<FileEntity>* field for picture.
* **Order:** 
  + Use our AddressEmbedded
  + Make the relationship to Shipper and Customer a *Lite<T>* relationship.

Finally, the only tricky point in Order entity is how to implement OrderDetails. It’s a relational table but has some information attached to the relationship (UnitPrice, Quantity, Discount).

In order to implement it, we will have to create an OrderDetailEmbedded:

***embeddedEntity*** [Tab] [Tab] OrderDetail [Enter]

***fieldLite*** [Tab] [Tab] ProductEntity [Tab] Product [Enter]

***field*** [Tab] [Tab] decimal [Tab] UnitPrice [Enter]

***field*** [Tab] [Tab] int [Tab] Quantity [Enter]

***field*** [Tab] [Tab] float [Tab] Discount [Enter]

The result should be like this after adding a ValidationAttribute and changing the base type.

[Serializable]

public class OrderDetailEmbedded : EmbeddedEntity

{

    public Lite<ProductEntity> Product { get; set; }

    [Unit("€")]

    public decimal UnitPrice { get; set; }

    public int Quantity { get; set; }

    [Format("p")]

    public decimal Discount { get; set; }

}

### PropertyValidation

Let’s push the validation system a little bit. Suppose that we want to be sure that discount is something like 5%, 10%... 25%, always a multiple of 5%.

We don’t have a Validator attribute that fits these requirements, but we could create one just by creating a class that inherits from ValidationAttribute.

In this case, however, we will just override PropertyValidation method in the entity itself:

protected override string? PropertyValidation(PropertyInfo pi)

{

  if (pi.Name == nameof(Discount))

            {

                if ((Discount \* 100.0m) % 5.0m != 0)

                    return OrderMessage.DiscountShouldBeMultpleOf5.NiceToString();

            }

            return base.PropertyValidation(pi);

}

This method will be called for every property of the entity, and if it returns and string, the property value will be considered wrong. If everything is ok it should return null.

This technique has the advantage that we can take into account more than one property value to make our validation logic. In this case we will need to use Notify(()=>OtherProperty) to force the re-evaluation of the validation logic for the affected properties after changing the value.

Then, just by creating a field *MList<OrderDetailEmbedded>* in OrderEntity entity we will have the expected result.

[NoRepeatValidator]

public MList<OrderDetailEmbedded> Details { get; set; } = new MList<OrderDetailEmbedded>();

Tree Structures:

Northwind has not any tree structures but if you have, you can inherit from *TreeEntity* instead of *Entity* class.

This class is looks like the following:

[Serializable]

    public abstract class TreeEntity : Entity

    {

        [UniqueIndex]

        SqlHierarchyId route;

        [InTypeScript(false)]

        public SqlHierarchyId Route

        {

            get { return this.Get(route); }

            set

            {

                if (this.Set(ref route, value))

                    this.ParentRoute = value.GetAncestor(1).ToString();

            }

        }

        static Expression<Func<TreeEntity, string>> RouteToStringExpression =

        @this => @this.Route.ToString();

        [ExpressionField]

        public string RouteToString

        {

            get { return RouteToStringExpression.Evaluate(this); }

        }

        [NotNullable, SqlDbType(Size = 255, SqlDbType = SqlDbType.VarChar)]

        public string ParentRoute { get; set; }

        static Expression<Func<TreeEntity, int?>> LevelExpression = @this => (int?)@this.Route.GetLevel();

        [Ignore]

        int? level;

        [ExpressionField("LevelExpression"), InTypeScript(true)]

        public int? Level

        {

            get { return level; }

            set { level = value; }

        }

        protected override void PostRetrieving()

        {

            this.level = (int)this.Route.GetLevel();

        }

        [Ignore, ImplementedByAll]

        public Lite<TreeEntity> ParentOrSibling { get; set; }

        [Ignore]

        public bool IsSibling { get; set; }

        [SqlDbType(Size = 255)]

        [StringLengthValidator(AllowNulls = false, Min = 1, Max = 255)]

        public string Name { get; set; }

        [NotNullable, SqlDbType(Size = int.MaxValue)]

        [StringLengthValidator(AllowNulls = true, Min = 1, Max = int.MaxValue)] //Set by BL

        public string FullName { get; private set; }

        public void SetFullName(string newFullName)

        {

            this.FullName = newFullName;

        }

        static Expression<Func<TreeEntity, string>> ToStringExpression = e => e.Name;

        [ExpressionField]

        public override string ToString()

        {

            return ToStringExpression.Evaluate(this);

        }

    }

    [AutoInit]

    public static class TreeOperation

    {

        public static readonly ConstructSymbol<TreeEntity>.Simple CreateRoot;

        public static readonly ConstructSymbol<TreeEntity>.From<TreeEntity> CreateChild;

        public static readonly ConstructSymbol<TreeEntity>.From<TreeEntity> CreateNextSibling;

        public static readonly ExecuteSymbol<TreeEntity> Save;

        public static readonly ExecuteSymbol<TreeEntity> Move;

        public static readonly ConstructSymbol<TreeEntity>.From<TreeEntity> Copy;

        public static readonly DeleteSymbol<TreeEntity> Delete;

    }

As you can see in the code when your entity inherited from this class of the framework, you will have properties and operations ready to work without any extra efforts, but actually you can add or even override every operation that you want. We will explain the mechanism in `Logic` document.

# MixinEntity

MixinEntity is the base class to create **Mixins**, an alternative to inheritance to expand types.

Mixins are used to append fields, properties, columns or methods to types in two important use-cases:

* When you don't have control the type because it's in a reusable module. (i.e.: Add a property EmployeeEntity in UserEntity only for this application).
* When you need to add properties to different types independently of their position in the inheritance hierarchy (i.e.: AddressEntity, OrderEntity and EmployeeEntity all need to be Corrupt, or Isolated, or Disconnected).

Under the covers a MixinEntity inherits from ModifiableEntity and is ensured to be appended to any instance of types that include the Mixin.

## Mixins vs Inheritance

Mixins have two advantages over inheritance:

### Real type expansion

When you inherit from a type (i.e.: AnimalEntity) and add some new properties to it you are actually creating a new type, not modifying the original one. That's Ok if you want to create a hierarchy of types (i.e.: LionEntity) but if you want expand the type (i.e.: CustomAnimalEntity) you'll need to find all the points where the old type is instantiated (i.e.: new AnimalEntity) and replaced by the new type (i.e.: new CustomAnimalEntity). You could use a factory instead, but this doesn't play nicely with [object initializers](http://msdn.microsoft.com/en-us/library/bb384062.aspx).

With Mixins you just associate the type with your custom Mixin (i.e.: MixinDeclarations.Register<UserEntity, UserEmployeeMixin>()) and then every single instantiated animal will carry your mixin with it, no need to review all the instantiations.

### Multi-directional expansion

With inheritance you only have one base class. This is OK to model some simple hierarchies but commonly types expand in different unrelated dimensions.

For example, imagine that we have a SAAS application using **Isolation module** and some entities require a field with the current isolation.

Additionally, some entities can be used off-line in a boat, using **Disconnected module**, and require some fields to know who is the current owner.

And finally, some entities have been loaded from a legacy application and have some validations disabled, using corruption.

Using inheritance it will be a mess. For example, a LionEntity:

* should inherit from AnimalEntity?,
* or from IsolatedAnimalEntity?
* or from DisconnectedIsolatedAnimalEntity?
* or from CorruptedDisconnctedIsolatedAnimalEntity?

maybe AnimalEntity should inherit from CorruptedDisconnctedIsolatedEntityEntity? but what if GiraffeEntity doesn't need to be used off-line, neither isolated or corrupted...

Using mixin the problem is simple, LionEntity inherits from AnimalEntity, but also includes IsolationMixin, DisconnectedMixin and CorruptMixin.

## MixinEntity vs EmbeddedEntity

The implementation of MixinEntity is quite similar to EmbeddedEntity, in fact both inherit directly from ModifiableEntity but there are two important differences:

1. MixinEntity can be included in types you don't have control of (using static class MixinDeclaratons).
2. Only one MixinEntity instance of each type can be associated to an entity (i.e.: is not possible to have two IsolationMixin), while the same is not true for EmbeddedEntity fields, that have a name (i.e.: ShippingAddress and BillingAddress).

## Declaring a new Mixin Type

Let's see how to declare a new Mixin type:

[Serializable]

public class UserEmployeeMixin : MixinEntity

{

    protected UserEmployeeMixin(Entity mainEntity, MixinEntity next)

        : base(mainEntity, next)

    {

    }

    public Lite<EmployeeEntity>? Employee { get; set; }

}

As you see, a MixinEntity looks like a normal entity, with normal properties and fields and the same capabilities for validation, change tracking, change notifications for data-binding etc...

The only important difference is the constructor:

* The constructor should be protected to avoid client code instantiate any MixinEntity. An instance with Mixins is automatically instantiated with all their mixins and there's no way to get rid of them. They are effectively an expansion of the type.
* The constructor passes an Entity mainEntity to the base constructor, this value is stored in the MainEntity property in MixinEntity to let any Mixin have access the main entity.
* The constructor passes a MixinEntity next to the base constructor, the reason is that, in-memory mixins are stored as a linked list.

## Associating Types with Mixins

There's two ways of associating an entity with a mixin:

* The simpler one, if you have control of the type, is to use an attribute:

[Mixin(typeof(DisconnectedMixin))]

[Serializable, EntityKind(EntityKind.Main, EntityData.Transactional)]

public class OrderEntity : Entity

{

...

}

* But frequently you want to add mixins to entities that you can not modify, in this case just include this line **before including the entitiy in the schema**:

MixinDeclarations.Register<UserEntity, UserEmployeeMixin>();

**Note:** \*Signum.Windows also requires this information, but is automatically transited by the WCF webservice when started.

At this moment, Southwind entities should be able to produce a database quite similar to Northwind, let’s try:

## Southwind.Logic

Reload the project Southwind.Logic. This project contains the business logic that will run on the server and will be used by the **Web** interface, the **Windows** interface through a WCF service, the **Load** application and the **Unit** **Tests**.

We will get into this topic more deep in the next tutorial, for now let’s start simple.

In order to create the database we first have to tell the engine witch entities will get into the schema.

***logic*** [Tab] [Tab] Order [Enter]

The rest of the entities get automatically included by walking the dependencies of OrderEntity.

Also we need to call `OrderLogic.Start(sb, dqm);` in Starter.cs.

## Southwind.Load

Next step is reloading Southwind.Load. In this project we have already created a simple Console Application that we can use for manipulating the schema, loading data, or any other administration tasks.

The template already has a menu that allows us to create and synchronize the database. Let’s just go to SQL Management Studio, connect to localhost, and create a new database.

By convention the name of the database should be the same than the project, in this case ‘Southwind’, but you can change it in the connection string.

Once created, just mark Southwind.Load as startup project and run. Choose the first option, “New Database” by pressing “N” and… wow! The schema get’s created according to the entities.

## Summary

In this tutorial we have seen how to create entities using a few simple primitives like Entity, EmbeddedEntity, or using *MList<T>* and *Lite<T>* and adding fields properties and validation rules.

Signum Framework is designed top-down to promote a code-first approach, and trying to make it work on a legacy database is a pain due to some strict conventions (like having Id and *ToStr* in every entity).

Nevertheless, we have seen how the generated schema is simple and predictable and can be exploited by third-party tools easily.

In the next tutorial we will get deep into how to write the business logic, load legacy data and create the user interfaces, and how these conventions will simplify our code on the long run.