# Signum Framework Tutorials - Part 7: Operations

## About Signum Framework

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 Signum Extensions

## Signum Extensions in a set of vertical modules (database, business logic and user interface) built on top of Signum Framework to enrich any application made with this technology.

## About this series

In this series of tutorials 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, windows (WPF) and web (React) user interface, data loading and any other aspect worth to explain.

If you want to know more about the principles of Signum framework look at the previous tutorial:

* [Signum Framework Principles](http://www.codeproject.com/KB/linq/SignumFramework.aspx)
* Signum Framework Tutorials Part 1 – Southwind Entities
* Signum Framework Tutorials Part 2 – Southwind Logic
* Signum Framework Tutorials Part 3 – Southwind Load
* Signum Framework Tutorials Part 4 – Southwind Web
* Signum Framework Tutorials Part 5 – Southwind Windows
* Signum Framework Tutorials Part 6 – Southwind Authorization

In this tutorial we will see how to start the Operations module and use operations as the fundamental building box of our business logic.

## Introduction

In the last tutorial, we saw how the complex Authorization module could be seamlessly integrated with the rest of the application, as any other module in Signum Extensions, just by calling the *Start* method.

Most of the modules extend the features of your application by registering new entities and queries and by taking profit of some extension points provided by the framework, without a lof of code changes. Operations are a little bit exceptional in this case.

An operation is basically a method with some extra metadata registered in a repository. They are the fundamental building block for your business logic, by writing your business logic using operations you get some benefits:

* Automatic user interface: the registered operations for an entity are automatically exposed as buttons in the entity button bar, or the search dialog.
* Automatic log: Every time an operation is called an OperationLogEntity is created, keeping an historic of who was doing what, or it there was an error.

Also, because they are registered as data in the server, it’s easier to give them other uses, like alternative user interfaces or auto-generated help.

Operations are registered using an Enum as the key, it can be of any enum type, but as a convention we use MyEntityOperation as the type name. This key is called operationKey.

By using enums we get some benefit:

* We can define them in the Entities assembly, so Windows application have IntelliSense and type checking.
* We can use *EnumLogic* and *Enum<T>* to easily synchronize them with the database.

## Types of Operations

Operations can be registered independently, or can be registered as transitions in a graph of states.

When they are part of a graph they have to define a set of possible origin states (FromStates) and a target states.

There could be more than one graph for each type of entity, each one with different states. Also, all the operations of the graph share some common features, like the same way to know the state from the entity.

In order to declare graphs we inherit from *Graph<E, S>* (entity, state) and then we don’t have to repeat the entity type and the state over and over.

Example:

public class OrderGraph : Graph<OrderEntity, OrderState>

{

static void Register()

{

GetState = o => o.State;

… // Execute, Delete, Construct, ConstructFrom, ConstructFromMany here

}

}

## Declaring Operations

In order to defined and later invoke an operation, as well as customizing their appearance in the UI we need a way to name it. Other frameworks would just have used an string for it, but in Signum Framework we are obsessed with strongly-typed APIs, so we use OperationSymbol instead.

The different members of the OrderOperation static class serve as keys that can only be used to declare/execute the right type of operation. They will be automatically initialized and, once registered, synchronized in the database. The class should be defined in your Entities assembly and he members will be translatable.

Example.

[AutoInit]

public static class OrderOperation

{

    public static ConstructSymbol<OrderEntity>.Simple Create;

    public static ExecuteSymbol<OrderEntity> SaveNew;

    public static ExecuteSymbol<OrderEntity> Save;

    public static ExecuteSymbol<OrderEntity> Ship;

    public static ExecuteSymbol<OrderEntity> Cancel;

    public static ConstructSymbol<OrderEntity>.From<CustomerEntity> CreateOrderFromCustomer;

    public static ConstructSymbol<OrderEntity>.FromMany<ProductEntity> CreateOrderFromProducts;

    public static DeleteSymbol<OrderEntity> Delete;

}

## Execute

Execute are the most common operation types, they just do something over an entity (Send Order, Cancel Order), and define the ‘from states’ and target state. It has the following members:

* **Execute:** An Action to be executed when the operation is invoked.
* **CanExecute:** A Function that returns whether a method could be executed in the current state of the entity or not. It there’s a problem returns and string explaining it, otherwise null.
* **AllowsNew:** A bool controlling whether the operation can be executed over new entities or not. By default not.
* **Lite:** When true, the database version of the entity is taken, otherwise the user entity is used (possibly with some changes).
* **FromStates:** (Execute only)the states from with the operation can be executed.
* **ToStates:** (Execute only) the states the entity should be at the end of the execution.

After the execution the entity it’s saved and a new LogOperationEntity created.

In the user interface, execute operations typically are shown as buttons on the toolbar of the entity itself. When CanExecute returns some error, then the button is disabled and the error is shown ‘OnMouseOver’. When the operation is not authorized however, the button is just not sown.

In the future, possibly, they could also be shown as a contextual menu on any row on a search window.

Example:

new Execute(OrderOperation.Ship)

                {

                    CanExecute = o => o.Details.IsEmpty() ? "No order lines" : null,

                    FromStates = { OrderState.Ordered },

                    ToStates = { OrderState.Shipped },

                    Lite = false,

                    Execute = (o, args) =>

                    {

                        o.ShippedDate = args.TryGetArgS<DateTime>() ?? DateTime.Now;

                        o.State = OrderState.Shipped;

                    }

                }.Register();

Manual Invocation:

orderLite.ExecuteLite<OrderEntity>(OrderOperation.Ship);

order.Execute<OrderEntity>(OrderOperation.Ship); //Is Order is not modified

## Construct

Just constructs the entity, optionally saving it to the database. If the entity is saved, then an OperationLogEntity will be created.

It has just a few members:

* **Constructor:** A function that returns a new o
* **ToStates:** (Construct only) the states the entity should be at the end of the construction.

From the user interface, it integrates seamlessly with the usual way of constructing objects (green plus symbol). In the case of more than one constructor being defined, the default behavior is to show a dialog and let the user choose the constructor (uncommon).

Example:

new Construct(OrderOperation.Create)

                {

                    ToStates = { OrderState.New },

                    Construct = (args) =>

                    {

                        var customer = args.TryGetArgC<Lite<CustomerEntity>>()?.Retrieve();

                        return new OrderEntity

                        {

                            Customer = customer,

                            ShipAddress = customer?.Address.Clone(),

                            State = OrderState.New,

                            Employee = EmployeeEntity.Current,

                            RequiredDate = DateTime.Now.AddDays(3),

                        };

                    }

                }.Register();

Manual Invocation:

OperationLogic.Construct<OrderEntity>(OrderOperation.Create);

## ConstructFrom

*ConstructFrom*, instead of building an entity out of nothing (or some optional parameters) assumes that the entity is constructed from another entity. With this extra information we can place a button on the original entity window.

*ConstructFrom* could be confusing because is an operation that lives between two different entity types. It’s declared in the graph of the target entity, but is shown as a button in the original entity, and shown as an operation belonging to the original entity in the authorization menu.

It has these members:

* **Construct:** Function that takes the original entity as a parameter, and some optional extra arguments, and returns the target entity as result.
* **CanConstruct:** Function that, given the original entity, returns whether is possible to construct the entity (returning null) or the reason why is not possible in a string.
* **AllowsNew:** A bool controlling whether the operation can be executed over new entities or not. By default false.
* **Lite:** When true, the database version of the original entity is taken, otherwise the user version of the original entity is used (possibly with some changes).
* **ToStates:** (ConstructFrom only) the state the entity should be at the end of the construction.

Example:

new ConstructFrom<CustomerEntity>(OrderOperation.CreateOrderFromCustomer)

                {

                    ToStates = { OrderState.New },

                    Construct = (c, \_) => new OrderEntity

                    {

                        State = OrderState.New,

                        Customer = c,

                        Employee = EmployeeEntity.Current,

                        ShipAddress = c.Address,

                        RequiredDate = DateTime.Now.AddDays(3),

                    }

                }.Register();

Manual Invocation:

miCustomerLite.ConstructFromLite<OrderEntity>(OrderOperation.ConstructFromCustomer);

miCustomer.ConstructFrom<OrderEntity>(OrderOperation.ConstructFromCustomer); //If miCustomer is not modified

## ConstructFromMany

*ConstructFromMany*, instead of being built from just one item, it’s build from a list of similar ones. It’s usually invoked from a menu in the original entities’ search window.

*ConstructFromMany*, just as *ConstructFrom* lives between two different entity types. It’s declared in the graph of the target entity, but is shown as a button in the original entity search window, and shown as an operation belonging to the original entity in the authorization menu.

It has these members:

* **Construct:** Function that takes a list of lites of the original entities as a parameter, and some optional extra arguments, and returns the target entity as result.

Example:

new ConstructFromMany<ProductEntity>(OrderOperation.CreateOrderFromProducts)

                {

                    ToStates = { OrderState.New },

                    Construct = (prods, args) =>

                    {

                        var dic = Database.Query<ProductEntity>()

                            .Where(p => prods.Contains(p.ToLite()))

                            .Select(p => new KeyValuePair<Lite<ProductEntity>, decimal>(p.ToLite(), p.UnitPrice)).ToDictionary();

                        var customer = args.TryGetArgC<Lite<CustomerEntity>>()?.Retrieve();

                        return new OrderEntity

                        {

                            Customer = customer,

                            ShipAddress = customer?.Address.Clone(),

                            State = OrderState.New,

                            Employee = EmployeeEntity.Current,

                            RequiredDate = DateTime.Now.AddDays(3),

                            Details = prods.Select(p => new OrderDetailEmbedded

                            {

                                Product = p,

                                UnitPrice = dic[p],

                                Quantity = 1,

                            }).ToMList()

                        };

                    }

                }.Register();

Manual invocation:

OperationLogic.ConstructFromMany<ProductEntity, OrderEntity>(listOfProductLites, OrderOperation.ConstructFromProducts);