# Signum Framework Tutorials Part 6 – Authorization

## About Signum Framework 2.0

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 React
* Signum Framework Tutorials Part 5 – Southwind Windows

In this tutorial we will show how easy is to add existing modules to an already build application, and we will explain what maybe is the most powerful module in Signum Extension: Authorization.

### Introduction to Signum Extensions (Vertical Modules)

Probably, the very best feature of Signum Framework compared to other ORMs is how simple it is to create vertical modules.

In most of the software out there, reutilization happens mostly in the very high level of the stack, the user interface. You can buy a set of nice user controls but you have to connect them to your proprietary database. Also most of your business logic tied to your concrete database, so it’s not easy to reuse. Re-using database tables from one project to another is something nobody is doing.

In Signum Framework the entities are just simple C# classes, not tied to a concrete database or schema, so that you can put them a library and reuse it in another database just by including them in the schema.

A vertical module is a set of entities, business logic, and user interfaces that, once Started, registers whatever is necessary to integrate with the rest of the application:

* Includes the entities in the database schema
* Registers associated dynamic queries
* Defines business logic
* Attaches to the necessary events in the schema or the user interface
* Registers user interface views

Since the module is self contained, starting it is as easy as calling the Start method in the server and in the client application. On the other side since they use the standard way of writing entities, queries and user interfaces, the module integrates seamlessly with the rest of the applications, with no artificial boundaries on the user interface, queries, etc..

From a deployment point of view, typically every module contains a just few entities (1..5), one or two logic classes, some WPF controls for a windows user interface, and some views and controllers for web applications.

It doesn’t make sense to have four different assemblies (Entities, Logic, Windows and Web) for just a few classes, so we usually place some modules together sharing the same assembly. In Signum.Entities.Extensions.dll there are the entities of different technical modules: Authorization, Operations, Processes, Charting…

However, the modules try to stay independent from each other and if you don’t start some module, it doesn’t clutter your user interface or database, and affects performance in any way.

## Authorization

The authorization module is one of the most complex ones, but it defines basic entities, like UserEntity, that are necessary for the rest of the modules, get prepared.

### AuthLogic: The Core

The core of the authorization module defines two main entities, UserEntity and RoleEntity. As any other module, it also contains an AuthLogic static class that contains the bussines logic and a *Start* method that register the necessary entities, queries and handles events.

There’s also another two AuthLogicClient classes, on in *Signum.Windows.Extensions* that registers the necessary controls for the WPF user interface, and another one AuthClient.tsx in *Signum.React.Extensions* wich registers React Components and Server Controllers.

A UserEntity contains a user name, a password hash, and email and just one RoleEntity. Roles however can contain sub-roles, inheriting from them their permissions. This way you don’t configure permissions to users that can change but to roles.

AuthLogic contains the logic that keeps the RoleEntity relationship as a Directed Acyclic Graph.

It also contains some method and properties for globally enabling and disabling the authorization in a region or code, or the whole application.

* *AuthLogic.GloballyEnabled*: Gets or sets whether the authorization rules are taking effect in the application as a whole.
* *AuthLogic.Disable*: Returns a IDisposable and disables authorization rules temporally in a region of code.
* *AuthLogic.IsEnabled*: Gets whether the authorization rules are taking effect (taking into account globally and temporally options).

Finally, it also enables features

### Authorization sub-modules for logging-in

The AuthLogic also contains the basic business logic for log-in and log-out, change password, etc… for more advanced features there are some extension modules:

#### ResetPasswordRequestLogic:

In web applications is usual to have a ‘remember password’ feature. This module enables this feature creating ResetPasswordRequestEntity entity with a random code and sending an email to the user with this code so the user can change the password after clicking in the link.

#### UserTicketLogic:

In web applications is usual to have a ‘remember me’ feature. This module enables this feature creating a UserTicketEntity entity that stores a code that can be used to login the next time. This code is stored in a cookie and will be updated every time the user logs-in. By default the tickets expire after 60 days and users can have up to 4 simultaneous tickets (for different computers or browsers).

#### PasswordExpiresLogic:

Enable a default behavior to set global password expiration intervals, so when the user log-in receives an alert if it’s close to expire, or forces a password change after login if it’s already expired.

### Authorization Rules sub-modules

AuthLogic core, by itself, is not able to authorize the access to any resource. There are many different sub-modules to enable or deny access to Types of entities, Properties, Dynamic Queries, etc...

Once started, these modules register themselves in the appropriate extension points of the framework so everything works with **no code changes** in most of the situations.

**Internal implementations:**

All this modules store their authorization rules in a similar way, in fact internally they all use the same code, AuthCache class, with 5 generic parameters, that provides them with the following capabilities:

* Store the Authorization rules in the database for any kind of code
* Cache the authorization rules to avoid accessing the database once the application is started.
* Provide the infrastructure to have a common user interface to mange permission, showing which rules are inherited and which ones overridden.
* An API for modify rules at runtime (for loading applications mainly)
* Import / Export all the authorization rules in a XML format to move them to another database.

#### Authorization rules algorithm

Any kind of authorization rule contains three things:

* The role its applied for
* The resource it is applied (TypeEntity, PropertyEntity, QueryEntity…)
* A value that indicates the access, depending on the resource (a bool, [Read, Write, None]…)

For any given role and resource, the access information have to be returned, but not all the rules are actually stored in the database, most of them are inherited or just deduced from the default values. Let’s see in more detail how this algorithm works with an example:

We have four roles:

* Animal
* Human inheriting from Animal
* Spider inheriting from Animal too
* Spider-Man inheriting from Animal and SpiderMan at the same time

And we have two permission:

* MakeSpiderWeb
* EatFlies
* DriveCar

We want Animals no to have any of these permissions, Spider to be allowed to MakeSpiderWeb and EatFlies, Humans to DriveCar and SpiderMan to inherit everything from Spider and Human, but deny EatFlies.

The following diagram will show witch authorization rules will be shown as overridden (and thus, stored in the database) and with one will be just deduced:

Spider Man

Spider

Man

Animal

MakeSpiderWeb

EatFlies

DriveCar

(overriden)

MakeSpiderWeb (overriden)

EatFlies (overriden)

DriveCar

(overriden)

MakeSpiderWeb (overriden)

EatFlies (overriden)

DriveCar

MakeSpiderWeb

EatFlies (overriden)

DriveCar

As we see, we had to override the rules to deny everything for animal. By default the rule system is permissive and will give you everything.

Also, in the case of conflict of multiple inheritance, SpiderMan was getting the maximum permission between Man and Spider, and we had to override EatFlies to resolve the conflict.

The algorithm also allows changing this ‘permissive by default’ to ‘restrictive by default’ for each Role and type or Resource using the Min/Max strategy:

* Max strategy: If there are no inheriting roles you have maximum access for all the resources, and in the case of conflict the maximum is selected.
* Min strategy: If there are no inheriting roles you have the minimum access for all the resources, and in the case of conflict the minimum is selected.

In both cases, if the role has just one inheriting role (Man or Spider), the strategy is indifferent since the access for all the resources is just inherited.

Let’s suppose we want Min strategy for Animal, so we don’t have to manually establish everything to false (important when mores resources can appear as the application evolves), but a Max strategy for SpiderMan, since is a super hero and by default we want it to be able what a human does, union what a spider does.

Spider Man

Spider

Man

Animal

MakeSpiderWeb

EatFlies

DriveCar

(overriden)

MakeSpiderWeb

EatFlies

DriveCar

MakeSpiderWeb (overriden)

EatFlies (overriden)

DriveCar

MakeSpiderWeb

EatFlies (overriden)

DriveCar

Min

Min

Max

Max

Now, the permissions in Animal are not overridden anymore, since the Min strategy is selected. Setting Min or Max for Man and Spider had no effect, and Spider Man user the Max strategy, just as before, so we have to override EatFlies so is denied.

Not that we know how the authorization rules work, and how AuthCache manage to store as few as possible in the database, and before getting into the different sub-modules for authorization rules over different resources let’s clarify the concept of EnumEntity.

##### **SymbolEntity and SymbolLogic<T>**

Enums are a common data-type to define a fixed amount of different values. Under the cover they are just numbers, but with a special superpower: Calling ToString on an enum value returns the name of the field where is was declared, not the underlying int. Enum values always remember their type and the field where they were declared.

public enum c  
{  
 Blue = 0,  
 Green,  
 Red,  
 Yellow,  
}

var c = Color.Blue;

c.ToString(); //returns Blue

Unfortunately, enums are not very flexible : There’s no way to add new values to an enum already declared in a third-party library. When this is necessary, the typical solution is to use magic strings instead of enums. But strings are not strongly typed: no auto-completion, no rename, no compile-time errors…

Symbols solve this problems. You can declare a new symbol type just by inheriting from Symbol, but you can declare new symbol fields/instances in any static class. Moreover, they inherit their name and type from the field where they were declared, just like enums.

[AutoInit]  
public static class MyFileTypes  
{  
 public static FileTypeSymbol AttachedImage;  
}

var c = MyFileTypes.AttachedImage;  
c.ToString(); //returns MyFileTypes.AttachedImage

Also, symbols are entities, so can be reference by other entities easily and with referential integrity. Moreover, the schema synchronizer understand and synchronizes the used symbols.

This features make symbols a replacement for magic strings that are as easy to use as enums and can be referenced in the database. They are simple read-only entities that are usually related to a piece of code in a dictionary, creating a strongly-typed bridge between code and data. (e.g.,Operations, Permissions, Process algorithms, etc... are all symbols).

#### PermissionAuthLogic/Client

**Resource:** PermissionSymbol(represents and custom action that can be authorized or denied)

**Access:** bool (the permission could be allowed or not)

**Notes:**PermissionAuthLogic defines method for registering some enums (values or types) as permissions (using *SymbolLogic<PermissionSymbol>* internally).

It also defines two extension methods over PermisionSymbol to test if some permission is allowed for the role of the current user (IsAuthorized) or for asserting it, throwing an UnautorizhedAccessException if is not (Authorize).

#### OperationAuthLogic/Client

**Resource:** OperationSymbol(represents an action that can be done over an entity)

**Access:** OperationAllowed (None / DbOnly / Allow)

**Notes:**It uses the *EnumLogic<OperationEntity>* defined by OperationLogic (next chapter) to keep the operationKeys and the OperationEntity entities synchronized. When started, it automatically register himself so no operation is executed if is not allowd.

On the user interface, the buttons disappear (instead of being disabled) when they are not allowed, both in the edit window, as in the search control.

Also, when administrating rules, the operations are grouped by entity to make it easier to manage, but in the case of a polymorphic operation, the same key is used for all the operation implementations.

#### QueryAuthLogic/Client

**Resource:** QueryEntity(represents a queryName object)

**Access:** QueryAllowed (None / EmbeddedOnly / Allow)

**Notes:**It uses QueryLogic to synchronize the queryName objects with QueryEntity entities. This class also keeps the QueryEntity table small, making lazy creation of the entities when necessary.

QueryAuthLogic registers itself in the *DynamicQueryManager* so it throws exception if is not allowed then in every attempt to be executed. It also provides some methods to check whether a queryName is allowed or not.

On the user interface, disables the menu options that are related to disabled queries, and the search buttons in the EntityControls.

Also, when administrating rules, the queries are grouped by the type of the entity column to make it easier to administrate, but in the case of a query which has an ImplementedBy entity column, the same key is used.

#### PropertyAuthLogic/Client

**Resource:** PropertyEntity(represents a PropertyRoute, starting from a concrete type, a sequence of properties that point to database column, more or less)

**Access:** PropertyAllowed (the property can be hidden [None], visible [Read] and writable [Modify]).

**Notes:**It uses PropertyLogic to synchronize the PropertyRoutes with the PropertyEntity entities. This class keeps the Property table small, making lazy creation of the entities when necessary.

PropertyAuthLogic register himself into the *DynamicQueryManager* to avoid returning columns that are related to disabled PropertyRoutes, whether it is a trivial relationship, or a complex one with a formula involving many of them.

On the user interface, the EntityControls (ValueLines, EntityLines, etc..) are made read only, or hidden, through the Task pipeline available in windows and web.

When administrating, the properties are grouped by concrete type and the rules on one property do not affect other properties of other type, even if they are in the same inheritance hierarchy and have the same name.

#### TypeAuthLogic/Client

**Resource:** TypeEntity (represents the Type of non-abstract entity included in the schema).

**Access:** TypeAllowed (the type can be hidden [None], visible [Read] and writable [Modify] and creable [Create] both in the User Interface and the Database).

**Notes:**It uses TypeLogic (in Signum Framework) to synchronize the Types with the entities.

TypeAuthLogic register himself into the engine, auditing retrievals, savings, and queries. It also provides methods to get the access for a Type, for the current role or other one.

In the user interface, it registers himself into the Navigator extension points to avoid viewing disabled entities, and disables the necessary buttons.

When administrating it, different configurations can be set for Database and UserInterface by shift clicking in the radio button; in this case the higher one is used as Database)

#### TypeConditions

TypeAuthLogic also allows to configure a custom TypeAllowed for a subset of rows of a table, instead of for the whole table.

The way to define this subset is to register a TypeCondition. A expression lambda that will be used to classify entities belonging to a group.

TypeConditionLogic.Register<OrderEntity>(SouthwindGroup.UserEntities, o => o.Employee == EmployeeEntity.Current);

TypeConditionLogic.Register<EmployeeEntity>(SouthwindGroup.UserEntities, e => EmployeeEntity.Current.RefersTo(e));

TypeConditionLogic.Register<OrderEntity>(SouthwindGroup.CurrentCustomer, o => o.Customer == CustomerEntity.Current);

TypeConditionLogic.Register<PersonEntity>(SouthwindGroup.CurrentCustomer, o => o == CustomerEntity.Current);

TypeConditionLogic.Register<CompanyEntity>(SouthwindGroup.CurrentCustomer, o => o == CustomerEntity.Current);

This is typically used to automatically filter the results of queries as a global policy, but can also be applied to make entities that are readonly when belonging to a particular group.

Just as for the whole table, the custom TypeAllowed values for a TypeConditoin registers themselves in the Database and the Navigator extension points to avoid retrieving/viewing/saving disabled instances.

When administrating it, different configurations can be set for Database and UserInterface by shift clicking in the radio button; in this case the higher one is used as Database).

For each table/entity, more than one type condition rule can be applied. In those cases they will be applied from the last one to the first one, finishing in the TypeAllowed of the table itself (fallback).

There are no union/intersection mechanisms for type conditions. If necessary you will need to declare a new type condition that expresses the intersection.

## Conclussion

In this tutorial we have seen the capabilities of the authorization modules. An unobtrusive and powerful authorization system built on top of Signum Framework.

The core is based in Users and Roles, with some extra sub-modules to handle password expires, and simplify some common web scenarios like ‘remember me’ or ‘I forgot my password’.

The real power comes from the Authorization rules, that allow creating a hierarchy of overridden rules to allow or deny access to any kind of resource in the application: Types, Properties, Queries, Operations…

Technically, the two brightest points is the ability to hide any field for any role that will automatically affects the queries, or being able to filter some sets of instances for a role, affecting any query or database operation.