# Signum Framework

# Entity Relationships

Given two entities A, and B, we will explain the different ways of making 1-1, 1-M and M-N relationships, the Pros and Cons and an example for each.

# One to One

There’s really one way of making this relationship: making a reference from one entity to the other, and put a **UniqueIndex over the reference field**. We have however some choices:

##### **Making it a big reference or a Lite.**

As usual, it depends on performance and validation constraints. If A needs date from B to be valid, or the user is going to retrieve data from A all the time, I will make it a big reference, otherwise a Lite.

##### **Making the reference from A🡪 B or from B🡪A.**

I would choose the ‘bigger’ entity to have the reference, since it will be less affected by performance and more meaning for the user. Let’s assume it’s A 🡪 B.

Since most of the time a 1 to 1 relationship is a ‘Conceptual entity’ divided in two tables, soon or later B wants to reach the A, there are some options:

* **Reference - Query Expression:** Create an extension from B to A.
  + Pros: No redundancy on the DB.
  + Cons: You can only write this code on the Logic assembly, validation is harder.
  + Cons: Requires another DB query to retrieve.
  + Example: UserDN 🡪 EmployeeDN / Employee.User()
  + Natural UI: EntityLine - QuickLink
* **Double reference:** Make a reference from A🡪B and another one from B🡪A. On, the other of neither can be Lites.
  + Cons: Redundancy in the Database.
  + Pros: If there are different subclasses of A, the reference B🡪A ensures 1-1 relationship.
  + Example: (PaymentDN,ChargeDN,BillDN) 🡪 AccountingEntryDN   
    AccountingEntryDN.Reason
  + Natural UI: EntityLine - EntityLine
  + **Reference – Reference**
    - Pros: Easy moving from A to B and from B to A. Easy Validation.
    - Cons: Retrieving B is not two times as expensive.
  + **Reference – Lite**
    - Pros: Easy moving From A to B. Easy validation of A.
    - Cons: Not that easy moving from B to A, and validation of B.
    - Trick: On PostRetrieving of A, you can load the back Lite reference.
  + **Lite – Lite**
    - Pros: Little performance overhead.
    - Cons: Not that easy moving from A to B or B to A. Not that easy validation.
* **Reference – Ignored:** There’s no FK from B to A, but on memory an ignored pointer from B 🡪 A is set.
  + Pros: Easy validation of A and B as a whole.
  + Pros: Allows the implementation of the **Extension pattern**.
  + Cons: You shouldn’t retrieve B by itself.
  + Trick: On PostRetrieving of A, you can load the ignored reference from B🡪A.
  + Natural UI: EmbeddedControl /DataBorder with AutoChild – Nothing

# One to Many – Many to One

### Let’s suppose we have many A for each B.

### ****Reference from A 🡪 B****

Usually A is a live entity, from a table where multiple row, inserts and updates are expected, while B is a kind-of-something, a category etc… There are two options:

##### **Big reference**

* Pros: Validation of A can easily depend on B
* Cons: Retrieving A is now more expensive.
* Example: OrderLineDN 🡪 ProductDN
* Trick**:** Consider catching ProductDN table.
* Natural UI: EntityLine/EntityCombo – SearchControl

##### **Lite reference**

* Cons: Validation of A cannot depend on B; neither B fields should be shown on the UI.
* Pros: Retrieving A is cheap.
* Example: ProductDN 🡪 Lite<CategoryDN>
* Trick**:** If no changes on CategoryDN are expected at all, and there is going to be business logic depending on the category, consider using an **enum** instead o an entity.
* Natural UI: EntityLine/EntityCombo – SearchControl

#### ****MList<A> on B class****

If what we consider a live entity is the class B, then a collection from B to A could be a better option. There are three options:

##### **MList<Lite<A>> with UniqueIndex**

* Pros: Retrieving B is faster since no A would be retrieved.
* Pros:Other kind of entities can make a FK to A.
* Cons: Validation of B cannot depend on fields of A entity, neither it should be shown.
* Example: PrintingQueueDN 🡪 MList<Lite<DocumentDN>>
* Natural UI: EntityList – QuickLink

##### **MList<A> with UniqueIndex**

* Pros: Validation of B can easily depend on fields of A entity, and can be shown in the UI.
* Pros:Other kind of entities can make a FK to A.
* Cons: Retrieving B is slower since As have to be retrieved.
* Example: CarDN 🡪 MList<Wheel>, OrderDN 🡪 MList<OrderLineDN>
* Natural UI: EntityList / Grid – QuickLink

##### **MList<A> being A Embedded or just a Value**

* Pros: Validation of B can easily depend on fields of A entity, and can be shown in the UI.
* Cons: No one else can FK to As, since they have no Id
* Cons: Retrieving B is slower since As have to be retrieved, but we save one join from the previous case.
* Example: EmployeDN 🡪 MList<TelefonDN> or even MList<string>
* Natural UI: EntityList Master Detail – Nothing

#### ****Connector entity AB, with UniqueIndex on A.****

The last possibility is to create a new entity, ‘AB’ that has an UniqueIndex to A. This could be interesting if A is the live entity but the relationship from A 🡪 B is rarely used and we don’t want to pay for it each time we retrieve A.

* Pros: No performance penalty retrieving A (or B).
* Cons: Two joins necessary to go from A <- AB 🡪 B
* Cons:From the user point of view, connector entities are not ‘real’ entities and they should be carefully hidden on the UI. Usually a better entity name can be found.
* Example: PersonDN <= PersonCompanyDN 🡪 CompanyDN
* Natural UI: QuickLink – SearchControl. **Hide PersonCompanDN**
* Example Right: PersonDN <= EmployeeDN 🡪 CompanyDN
* Natural UI: QuickLink – EntitiLine, EntityLine - SearchControl.

# Many to Many

### Now we have a many to many relationship between A and B. Let’s see how to model it with Signum Framework.

#### ****MList<A> on B class****

### Assuming B is the live entity and/or B has fewer As than A has Bs. For example,

### Right: it makes sense that an Order has a collection of Products.

### Wrong: while a Product having a collection of Orders is a mistake.

In any case, both entities have ID and no one ‘belongs’ to the other one.

There are two possibilities:

##### **MList<Lite<A>>**

* Pros: Retrieving B is faster since no A would be retrieved.
* Cons: Validation of B cannot depend on fields of A entity, neither it should be shown.
* Example: PersonDN 🡪 MList<Lite<CountryDN>>

##### **MList<A>**

* Pros: Validation of B can easily depend on fields of A entity, and can be shown in the UI.
* Pros:Other kind of entities can make a FK to A.
* Cons: Retrieving B is slower since As have to be retrieved.
* Example: PackDN 🡪 MList<ProductDN>

##### **Different Entities**

Let’s first discuss what to do when A != B

##### **Connector Entity**

Another way to make a N..M relationship is to create an entity in the middle, dividing the relationship on 1..M + N..1

* Pros: No performance penalty retrieving A or B.
* Cons: Modifications on the UI are usually committed one by one, not when saving. Sometimes through operations.
* Cons:From the user point of view, connector entities are not ‘real’ entities and they should be carefully hidden on the UI. Usually a better entity name can be found.
* Example: PersonDN 🡨 PersonHouseDN 🡪 HouseDN
* Natural UI: SearchControl – SearchControl. **Hide PersonHouseDN**

##### **Entity in between**

Very often these hidden connector entities have some extra fields and/or a meaningful name for the user. Then the hidden entity comes out and becomes a proper entity, with name, UI, etc..

* Pros: No performance penalty retrieving A or B.
* Cons: Modifications on the UI are usually committed one by one, not when saving. Sometimes through operations.
* Example: PersonDN 🡨 AttendanceDN 🡪 EvenDN
* Natural UI: SearchControl – EntitiLine, EntityLine - SearchControl.

##### **Same Entities**

There are some special problems when A == B. In this case, once you are on an entity it’s not easy to know at which side of the relationship you should be and if you try both **SQL Server performance complains!**

##### **Signed Entity**

Even if both entities are from the same time, sometimes the entities are signed and the relationship can only be established by instances of opposite sign. If you keep one of the references for the positive ones and the other for the negative ones, you can save the face.

* Pros: No performance penalty retrieving A or B.
* Cons: Modifications on the UI are usually committed one by one, not when saving. Sometimes through operations.
* Example: AccountingEntryDN+ 🡨 ConciliationDN 🡪 AccountingEntryDN-
* Natural UI: SearchControl – EntitiLine, EntityLine - SearchControl.

##### **Duplicated relationship**

In case the relationship doesn’t follow any sign rules, under our experience, since trying both sides of the relationship has such impact on SQL Server performance, we better duplicate the relationship in both directions. Since there are two instances, better we hide them in the UI using Operations to modify it.

* Pros: No performance penalty retrieving A or B.
* Pros:Avoid performance penalty of trying both sides
* Cons: Double space on the db to store the relationship.
* Cons: Modifications on the UI are usually committed one by one, not when saving. Sometimes through operations.
* Example: PersonDN 🡨 FriendshipDN 🡪 PersonDN
* Natural UI: SearchControl - SearchControl.