Contents

You have to write a **custom id generator** when a new entity's permanent key must be calculated with information only accessible on the server. This topic covers how to write an implementation of the DevForce **IIdGenerator interface** for that purpose.

Overview

Every newly created entity requires a permanent unique key to be stored in the database. The key could be generated by the **database** or it could be assigned in creation code on the **client** (e.g, with a GUID). These are the simple cases covered in the entity creation topic.

Sometimes you have to calculate the key on the middle tier server. The database can't generate it and you can't determine it on the client because a critical ingredient in your key definition scheme is only accessible on the server. You'll have to write a custom id generator.

Your custom id generator is a class that implements the DevForce *IdeaBlade.EntityModel.IIdGenerator*. You deploy this class on both the server and the client and let DevForce discover it.

A scenario

Imagine that your application data are stored in a legacy relational database. The tables have integer key columns but key values are not store-generated ... and you are not allowed to modify the database so that they can be store-generated.

Instead, the database has a special table of ids called the “NextId” table. It has a single row and a single column holding the next available integer id. To get a new id, you'll have to read the current "next id" value, increment the "next id" value in the table, and save. Of course you'll have to worry about concurrency; you must prevent some other process from reading the same "next id" value or overwriting your "next id" value. So you either lock the table or perform these operations within a transaction.

You can't let remote clients perform this operation even if they could. The contention for that one precious "next id" value would be enormous. There are many potential clients. There is only one or maybe a few servers. We'll have to let the server(s) manage the "next id".

You put the code to manage the "next id" in your custom id generator class.

An example

Most tables in the NorthwindIB tutorial database that ships with DevForce are defined with store-generated integer ids. The **Territory** table does not. If you map a Territory entity to the Territory table, it will have an integer id property ("TerritoryID") that you'll have to manage with a custom id generator.

The NorthwindIB tutorial database also has a **NextId** table. The "Custom Id Code Generation" code sample [LINK] includes a custom id generator class that uses this table to calculate permanent ids.

You would not map the NextId table to a "NextId" entity nor use Entity Framework to maintain that table. You would use raw ADO.NET commands as illustrated in the code sample.

We refer to Territory and the NextId table throughout this topic.

Temporary ids

In principle, the client *could* ask your server-side id generator for the "next id" every time it created a new entity. That would be inefficient. It won't work at all if the client is disconnected from the server. DevForce is designed to work disconnected.

We have to be able to create new entities without accessing the server. The solution is to use temporary ids and replace them with permanent ids before save.

Accordingly, when you create a new entity, you give it a key with a temporary id. More precisely, you tell a DevForce EntityManager to give the new entity a temporary id as shown in this example of a static Create method for Territory.

C#
```
public static Territory Create(
 EntityManager mgr,
 string territoryDescription, int regionID)
{
 Territory newTerritory = new Territory();
 mgr.GenerateId(newTerritory, Territory.PropertyMetadata.TerritoryID);
 // other code
 newTerritory.EntityAspect.AddToManager();
 return newTerritory;
}
```

VB
```
Public Shared Function Create( _
 ByVal mgr As EntityManager, _
 ByVal territoryDescription As String, ByVal regionID As Integer) As Territory
 Dim newTerritory As New Territory()
 mgr.GenerateId(newTerritory, Territory.PropertyMetadata.TerritoryID)
 ' other code
 newTerritory.EntityAspect.AddToManager()
 Return newTerritory
End Function
```

The critical line is the *GenerateId* method of the EntityManager ("mgr") where we supply the new Territory entity ("newTerritory") and identify the property that gets the generated id.

```
mgr.GenerateId(newTerritory, Territory.PropertyMetadata.TerritoryID);
```
```
mgr.GenerateId(newTerritory, Territory.PropertyMetadata.TerritoryID)
```

We have to tell the *GenerateId* method which property to generate. The key could be a composite made up of multiple properties. DevForce can't guess which of those properties requires a temporary id. If the key consisted of two id properties, both of them custom generated, we'd have to make two calls to *GeneratId*, each identifying the property to set.

How does DevForce know what temporary id value to assign? DevForce doesn't know. Your custom id generator does. You specify how to create a temporary id in the *GetNextTempId* method of your generator. The DevForce *GenerateId* method calls your *GetNextTempId* and remembers that temporary id in the *TempIds* collection that you also implemented in your generator.

NorthwindIB Territory ids are always positive integers. A negative integer is a good candidate as a temporary id. We pick an arbitrary seed (such as -100) and decrement it each time we need a new id.

Let's suppose that your generator returned a temporary id of -101 and that the EntityManager's *GenerateId* method assigned -101 to the "newTerritory" entity.

The "newTerritory" entity can carry that value around as long as it stays on the client. We'll only have to replace it with a permanent id when we save it. More on that soon.

Temporary foreign key ids

NorthwindIB has an EmployeeTerritory table which associates Employees and Territories in a many-to-many relationship.

EmployeeTerritory has a "Territory" navigation property that returns the related Territory entity. It also has a "TerritoryID" property which holds the id of that related Territory entity. "TerritoryID" is a foreign key id ("FK Id") property.

Suppose we had an instance of EmployeeTerritory called "someEmpTer" and we set its "Territory" property with with "newTerritory".

```
someEmpTer.Territory = newTerritory;
```
```
someEmpTer.Territory = newTerritory
```

Now we have -101 in two places: once as the primary key of "newTerritory" and again in the FK id of "someEmpTer".

Everything is fine until we save. Before we can save, we'll have to replace the -101 values in both places with the permanent id for "newTerritory" ... an id value we don't have yet.

Id Fix-up

DevForce replaces all temporary id values with permanent values when we finally ask the EntityManager to save the changes. This process is called "Id Fix-up", and occurs during the save.

In the fix-up process:

- DevForce on the client sends the temporary ids to the EntityServer, along with all entities to be saved.
- The EntityServer asks your custom id generator's *GetRealIdMap* method to produce a dictionary of temporary/permanent id pairs.
- Your *GetRealIdMap* gets the permanent ids (perhaps from a NextId table as discussed earlier) and pairs them with the temporary ids, and returns this dictionary.
- The EntityServer uses this mapping of temp-to-perm ids to perform a fix-up on entities to be saved, and then performs the save.
- The EntityServer returns the temp-to-perm id map to the client, along with the saved entities.
- DevForce merges the saved entities into the client's cache, using the temp-to-perm id map for guidance.
- DevForce tells your client-side id generator to reset itself and clear the list of temporary ids.

You don't have to wait for save to fix-up the ids. You can call the EntityManager's *ForceIdFixup* (or *ForceIdFixupAsync*) methods any time.

Observations

The NextId table is a familiar example that comes in many variations. In NorthwindIB there is a single "next id" value for all entities that need custom id generation. Another database might have different "next id" values for each entity type.

Your application might not have a NextId table. It might have some other source of "next id" values. The commonality running through all custom id generators is that id calculation depends upon an external resource that can only be accessed on the server.

An application could have a mix of id generation strategies. Most of the NorthwindIB tables have auto-increment (identity) columns that don't require the attention of your custom id generator. Territory is the exception.

Perhaps you noticed that DevForce calls some id generator methods on the server and calls others on the client (and it calls a few members on both tiers). The specifics are detailed in the API documentation.

Most developers will write one class and deploy it to both the client and the server. The easiest way to deploy it is simply to include it in the same project as your entity classes; you'll deploy that project to both client and server anyway.

If the single-class approach is not to your liking, you can write two classes that implement the IIdGenerator interface, one for the client and one for the server. It's up to you to ensure that they are compatible.

You don't have to reference the custom id generator class in any of your projects. DevForce will discover it using its MEF-based extensibility mechanism.

Sample Id Generator

See the code samples for the source code of an example numeric id generator class that you can either use directly or adapt for your application.