**Contents**



Set the entity's unique *EntityKey* in the constructor unless it is store-generated or managed by a custom key generator.

# EntityKeys in brief

Every entity instances has an *EntityKey* that uniquely identifies and distinguishes it from every other entity. An entity's *EntityKey* is also a DevForce object that corresponds to the primary key in the mapped database table.

You can't put an entity in an EntityManager cache until it has an EntityKey value. No two entities with the same EntityKey value can coexist in an entity cache.

You can ask an entity instance for its EntityKey:

```
var key = someEntity.EntityAspect.EntityKey;
var id = key.values[0];
```

```
Dim key = someEntity.EntityAspect.EntityKey
Dim id = key.values(0)
```

An EntityKey instance holds one or more key values. Each is the value of a property of the entity. That property is called a "key property". It was defined during the mapping exercise when you built your EntityModel.

An entity class can have multiple key properties (a "composite key"); most entity classes only have one key property, typically called the "ID".

Whether it's an ID or a composite key, your objective as a creator of entity objects is to set the EntityKey as quickly as possible so that it can enter the cache and be displayed, modified and saved.

# You don't set auto-generated keys

You don't have to set the key if it is "store generated". An ID property mapped to an Identity column of a table in SQL Server is a typical example of a store-generated EntityKey. DevForce will initialize such a key for you.

In fact, you **can't** assign such a key and make it stick. DevForce will override your setting with a temporary value the moment you add the entity to an EntityManager.

Similarly, if the entity requires a custom id, you'll have to wait until you add the entity to an EntityManager before you initialize the key by calling *EntityManager.GenerateId(...)* as described in the "Generate custom ids" topic.

You definitely want to wrap this complexity in a factory method.

If you don't have a store-generated key or a custom generated key, you must set the key yourself and perhaps the best place to do that is in the constructor.

There are two ways to set the key: set it directly using a local key generator or set it using parameters passed into the constructor.

# Set a GUID key in the constructor

A **GUID** makes a fine key in part because it is easy to generate locally. You should set the GUID key in the constructor.

```
public Customer()
{
  ID = SystemGuid.GetNewGuid();
}
```

```
Public Sub New()
  ID = SystemGuid.GetNewGuid()
End Sub
```

You ask "What is *SystemGuid.GetNewGuid()*? There is no *SystemGuid* class in .NET !"

Right you are. *SystemGuid* is a utility class that you should consider adding to your application because it helps you control what kind of Guid you create and makes it easier to write repeatable tests of your entity creation logic.

## Set a key with constructor parameters

Sometimes the key for the new entity should be the same as the key of another entity. This typically happens when a root entity has an optional companion that extends the root with extra information.

The companion is a "dependent" entity in a one-to-one relationship with its parent "principal" entity. The companion's key is always the same as its parent's key.

The principal entity usually maps to one table in the database and the dependent entity to a different table. You could model them together as a single conceptual entity in Entity Framework. However, practical reasons may oblige you to model them as separate entities as we're doing in the following example.

Suppose we have a Customer entity and a CustomerExtra entity. Maybe the CustomerExtra carries a potentially giant image of the Customer. The Customer entity is lean and light so you can afford to retrieve lots of Customer entities. You don't worry about the cost of the image. You only need the image when the user drills-in for more detail about the customer; that's when you'll retrieve its companion CustomerExtra.

The Customer entity is the principal, CustomerExtra is the dependent, and you can navigate between them via their reciprocal reference properties. CustomerExtra's ID is always the same as its parent Customer's ID.

The CustomerExtra constructor needs one parameter, the parent customerID. It might look like this:

```
public CustomerExtra(int parentCustomerID)
{
  CustomerID = parentCustomerID;
}
```

```
Public Sub New(ByVal parentCustomerID As Integer)
  CustomerID = parentCustomerID
End Sub
```

## Set composite keys with constructor parameters

A "linking" entity associates two other entities in a many-to-many relationship. The Northwind OrderDetail entity has a key with two parts, a composite key.

One part is the ID of the parent Order. The other is the ID of the associated product, the product being purchased. the two parts of the key are simultaneously the foreign key ids supporting the navigation properties - OrderDetail.Order and OrderDetail.Product.

The OrderDetail has to have a complete composite key - both ids - before it can be added to an EntityManager. These ids can't be calculated; they are invariants supplied at the moment of creation.

The constructor could look like this.

```
public OrderDetail(int parentOrderID, int productID)
{
  CustomerID = parentCustomerID;
  ProductID = productID;
}
```

```
Public Sub New(ByVal parentOrderID As Integer, ByVal productID As Integer)
  CustomerID = parentCustomerID
  ProductID = productID
End Sub
```

## Consider entity parameters instead of ids

In the previous example, the constructor parameters were the ids of related entities. The caller of the OrderDetail constructor would probably extract the ids from the parent Order and Product and "new" an OrderDetail like so:

```
  var newDetail = new OrderDetail(anOrder.ID, aProduct.ID);
```

```
 Dim newDetail = new OrderDetail(anOrder.ID, aProduct.ID)
```

There are times when when this kind of constructor is useful ... even necessary. More often it is better to pass the entities rather than ids.

First, it's usually easier on the caller who has the Order and Product entities in hand. The caller doesn't have to bother with or even know about their idss; it just passes Order and Product.

```
var newDetail = new OrderDetail(anOrder, aProduct);
```

```
Dim newDetail = new OrderDetail(anOrder, aProduct)
```

And it lets the alternative constructor do the dirty work:

```
public OrderDetail(Order parentOrder, Product product)
{
  EnsureOrderAndProductAreGood(parentOrder, product);
  OrderID = parentOrder.ID;
  ProductID = product.ID;
}
```

```
Public Sub New(ByVal parentOrder As Order, ByVal product As Product)
  EnsureOrderAndProductAreGood(parentOrder, product)
  OrderID = parentOrder.ID
  ProductID = product.ID
End Sub
```

The more important reason to prefer passing entities is that it enables richer input validation. The "*Ensure...*" method can do more than check for null. It can validate the Order and the Product; make sure they're well matched; make sure that the Order is valid for this detail (and vice versa).

Lesson: prefer constructors with entity parameters.

## Postpone adding to manager

The constructor could have set the Order and Product reference navigation properties instead of setting the corresponding foreign key ids. The implementation might have been this:

```
public OrderDetail(Order parentOrder, Product product)
{
  EnsureOrderAndProductAreGood(parentOrder, product);
  Order = parentOrder;
  Product = product;
}
```

```
Public Sub New(ByVal parentOrder As Order, ByVal product As Product)
  EnsureOrderAndProductAreGood(parentOrder, product)
  Order = parentOrder
  Product = product
End Sub
```

But it wasn't. The author chose to set the ids instead of the navigation property because he didn't want to add a new OrderDetail to an EntityManager just yet.

The author understood that if he set the navigation property - the Order property in this case - DevForce would immediately **add** the OrderDetail to the EntityManager of the parent Order. That's the expected behavior described in the "Add, attach, and remove" topic. He did not want to add it to a manager.

Perhaps the author was hoping use this constructor in OrderDetail tests. Perhaps he planned to create design-time OrderDetail entities inside Blend. Testing and designing are easier if the new OrderDetail is **attached** to an EntityManager in the manner of a pre-exising entity rather than **added** as a new entity . He kept these options open.