Contents

Aside from the custom *EntityManager*, the rest of the generated source code file concerns the entity model and all most all of the file consists of **entity class** definitions. This topic offers an orientation to the code generated for the *Customer* class shown in this screenshot.

```
/// <summary>The auto-generated Customer class. </summary>
[DataContract(IsReference=true)]
[IbEm.DataSourceKeyName(@"NorthwindEntities")]
[IbEm.DefaultEntitySetName(@"NorthwindEntities.Customers")]
public partial class Customer : IbEm.Entity {

  /// <summary>Returns the property path for the given expression. </summary> ...
  public static string PathFor(System.Linq.Expressions.Expression<System.Func<Customer, object>> expr) ...

  Data Properties

  Navigation properties

  EntityProperty definitions

  EntityPropertyNames
}
```

Attributes

The attributes leap out at you. Those familiar with Window Communication Foundation (WCF) will recognize the *DataContract* attribute: DevForce-generated entities are ready for serialization … and not just by DevForce itself as it shuttles entities between server and client. You can serialize entities in your own way for your own purposes as well (e.g., to a file for safe keeping).

The *DataSourceKeyName* is a DevForce attribute that identifies symbolically the data source where the Customer entity data are stored.

IbEm is an alias for the DevForce *IdeaBlade.EntityModel* namespace. Namespaces appear repeatedly throughout the file; the alias helps reduce the file size.

Why would we mark each entity class with its data source? Why not denote the data source at the model level?

Because DevForce can mix entities from different data sources in the same container. We can mix entities from Database “A” and entities from Database “B” in the same *EntityManager* cache. We can save changes to those entities too; DevForce will arrange a distributed transactional save on the server.

DevForce *Entity* base class

The Customer entity inherits from the DevForce *Entity* class:

```
public partial class Customer : IbEm.Entity {...}
```

```
Partial Public Class Customer
 Inherits IbEm.Entity
  ...
End Class
```

All generated entity classes inherit (ultimately) from the *Entity* class whose primary functions concern change tracking and access of entity state.

The *Entity* base class is no serious threat to "testability" or "extensibility". You can "new" a Customer in test environments without creating or mocking a dependency. *Entity* itself does not have any persistence functions. It doesn’t query or save. It is persistence ignorant in this strict sense.

It may hold a reference to the EntityManager to which it is attached (if, in fact, it is attached); you can "new" a fully functioning and self-sufficient test *EntityManager*/ in a single line. True, your *Customer* is DevForce framework aware and your model project must refer to DevForce libraries. You need those references for a functioning system in any case.

You also can inject your own custom entity base class into the entity model inheritance hierarchy

Data properties

The properties defined within the *Properties* region provide access to persistent data. As such they are often called “data properties” to distinguish them from the other inhabitants of the property zoo.

Data properties are the .NET manifestations of the entity properties described in the CSDL section of the entity data model (EDM).

Each data property returns a single instance of a type, usually a simple .NET type such as string or int. A property that returns a simple .NET type can return null if the property is nullable. It could return a ComplexType - an encapsulation of multiple data values in a single type; *ComplexType* data properties never return a null value.

All data properties in this example return simple .NET types as illustrated by the *Customer.CompanyName* property:

```
[Bindable(true, BindingDirection.TwoWay)]
[Editable(true)]
[Display(Name="Company Name", AutoGenerateField=true)]
[IbVal.StringLengthVerifier(MaxValue=40, IsRequired=true,
   ErrorMessageResourceName="Customer_CompanyName")]
[DataMember]
public string CompanyName {
 get { return PropertyMetadata.CompanyName.GetValue(this); }
 set { PropertyMetadata.CompanyName.SetValue(this, value); }
}
```

```
<Bindable(True, BindingDirection.TwoWay), Editable(True), _
 Display(Name:="Company Name", AutoGenerateField:=True), _
 IbVal.StringLengthVerifier(MaxValue:=40, IsRequired:=True, _
 ErrorMessageResourceName:="Customer_CompanyName"), DataMember>
Public Property CompanyName() As String
 Get
  Return PropertyMetadata.CompanyName.GetValue(Me)
 End Get
 Set(ByVal value As String)
   PropertyMetadata.CompanyName.SetValue(Me, value)
 End Set
End Property
```

The profusion of attributes may seem disturbing. There are more attributes adorning the property than there are lines of code within it!

The attributes help the entity fulfill the multiple roles we expect it to play on both client and server. *Bindable*, *Editable*, and *Display* are hints to UI tooling that affect how user controls are chosen and configured. The *StringLengthVerifier* is a validation rule derived from constraints expressed in the CSDL. *DataMember* is WCF markup indicating that the value of this property should be serialized.

You can alter or remove all of the attributes except *DataMember* by suppressing them in the EDM designer and by adding alternative annotations in a metadata buddy class.

The property is public by default. You can make it less accessible in the EDM designer.

Caution: properties of entities used in Silverlight must be public or internal. Because Silverlight prohibits access to protected and private properties, DevForce couldn’t read or write the property when serializing to a Silverlight client.

The get and set implementations are one-liners that delegate to a member of the entity’s inner *PropertyMetadata* class. The *PropertyMetadata* class is an advanced topic beyond the scope of this overview. But we can infer some of its capabilities by looking at how it is called within a property definition.

```
 get { return PropertyMetadata.CompanyName.GetValue(this); }
 set { PropertyMetadata.CompanyName.SetValue(this, value); }
```

```
Get
  Return PropertyMetadata.CompanyName.GetValue(Me)
End Get
Set
  PropertyMetadata.CompanyName.SetValue(Me, value)
End Set
```

The property is relying on helper methods to perform functions that go beyond simple value access. In fact, each helper method establishes a property value pipeline – one going in and one going out – that can do more than access a backing field.

The setter's in-bound pipeline can intercept values, validate them, and raise events such as *PropertyChanged*. The getter's out-bound pipeline allows the developer to intercept values on their way out.

Property interception, validation, and change notification are vital DevForce features that grant the developer fine-grained control over the behavior of an entity at the property level. The simple, one-line syntax conceals the potential for extending the property behavior with custom business logic – logic that might be part of the class definition or might be injected dynamically from sources outside the class.

Navigation properties

A navigation property is a means to retrieve an entity (or a list of entities) to which this entity is related by an association. The association was defined in the Entity Data Model (EDM).

In this example, the Customer has an Orders property that returns a list of Orders. For a given customer, the Orders property returns the order placed with that customer.

When we exercise the Orders property we say we navigate from the customer to its orders.

Here is its implementation:

```
[Bindable(false)]
[Display(Name="Orders", AutoGenerateField=false)]
[DataMember]
[IbEm.RelationProperty("FK_Order_Customer", IbEm.QueryDirection.ToRole1)]
public IbEm.RelatedEntityList<Order> Orders {
 get { return PropertyMetadata.Orders.GetValue(this); }
 }
```

```
<Bindable(False), Display(Name:="Orders", AutoGenerateField:=False), _
  DataMember, IbEm.RelationProperty("FK_Order_Customer", IbEm.QueryDirection.ToRole1)>
Public ReadOnly Property Orders() As IbEm.RelatedEntityList(Of Order)
 Get
   Return PropertyMetadata.Orders.GetValue(Me)
 End Get
End Property
```

We have attributes as we did for the data properties.

One attribute in particular identifies both the association (called a relation in DevForce) and the direction of the association. That tells DevForce that you are navigating from the Customer to the Orders. More on Associations in a moment.

The Orders navigation property is significantly different from the *CompanyName* data property we reviewed earlier. *Orders* returns a list of entities, not a data value. The entities could come from the local cache or the get could trigger so-called lazy loading of orders from the server; whether the property does or does not lazy load is dynamically configurable.

A lazy load operation is performed asynchronously in both Silverlight and Windows Store applications, since the queries to the *EntityServer* must be done asynchronously. The property immediately returns a list of whatever related orders (if any) happen to be in cache; the list could be empty. Meanwhile, the attached *EntityManager* asks the server for the related orders in background. When the orders arrive, the *EntityManager* populates the list and the *PendingEntityListResolved* event is raised.

Notice that the get implementation delegates to the *PropertyMetaData* class just as the data property did. The *Orders.GetValue* is a pipeline the developer can control and intercept with logic of his own.
Notice that there is no set. The *Order* property is read-only. *Orders* is a collection navigation property, a navigation returning a list of entities. The developer can add and remove entities from the list but cannot replace the list itself.

There is another kind of navigation property - the reference navigation property – that returns a single entity. This particular Customer entity doesn't have a reference navigation. We suspect that Order does have a reference navigation property – one that navigates back from the order to its parent Customer.

We suspect such a property exists but we cannot be sure without looking. Navigation properties are optional. You can choose to omit navigation properties from code generation. The association between Customer and Order would remain even if you opted out of both navigation properties.
In this case, Order does have a Customer property.

Here is Order's *Customer* property

```
[Bindable(false)]
[Display(Name="Customer", AutoGenerateField=false)]
[DataMember]
[IbEm.RelationProperty("FK_Order_Customer", IbEm.QueryDirection.ToRole2)]
public Customer Customer {
```

```
  get { return PropertyMetadata.Customer.GetValue(this); }
  set { PropertyMetadata.Customer.SetValue(this, value); }
 }
```

```
<Bindable(False), Display(Name:="Customer", AutoGenerateField:=False), _
  DataMember, IbEm.RelationProperty("FK_Order_Customer", IbEm.QueryDirection.ToRole2)>
Public Property Customer() As Customer
 Get
   Return PropertyMetadata.Customer.GetValue(Me)
 End Get
 Set(ByVal value As Customer)
   PropertyMetadata.Customer.SetValue(Me, value)
 End Set
End Property
```

It has both a get and a set. That means we could set the parent Customer as well as retrieve it.

The get could trigger a lazy customer load if the parent customer is not in cache.

DevForce will also lazy load a reference navigation property asynchronously in Silverlight and Windows Store applications. The property immediately returns a special form of the Customer entity instance called a *PendingEntity* . This instance has placeholder values until the actual customer data arrive from the server.

A reference navigation property always returns a value, never a null. If an order has no parent customer, the property returns a special Customer entity called the *Null Entity* (aka, nullo). The null entity reduces the likelihood of null reference exceptions and eliminates much of the null value checking that plagues the typical code base.

The null entity has permanent placeholder values; all of its reference navigation properties return null entities as well so you can have long chains of navigations (*order.Customer.SalesPerson.Name*) without incurring a null reference exception. You can always as an entity if it is the null entity.

Finally, the promised word about Associations. Associations (Relations) are inherently bidirectional. A Customer has Orders; each Order has a Customer. When defining a navigation property we have to say which way we're going: from customer to orders or from orders to customer. The Role (either Role1 or Role2) indicates the direction.

You can usually guess which entity type is Role1 and which is Role2 by the name of the *RelationProperty* (e.g., *FK_Order_Customer*). To be certain, you must look at actual definition of that *RelationProperty* in the generated class called *EntityRelation*. You'll always find the *EntityRelation* class at the bottom of the generated code file.

DevForce supports the same cardinality of Associations as Entity Framework. They can be 1–1, 0..1– 1, 1– 0..1, 0..1– M, M– 0..1, and M– M.