Creating a Powerful Business Objects Data Access Layer to Your MultiValue Database

BY DAVID COOPER, LEAD DEVELOPER, BLUEFINITY INTERNATIONAL

In this article we will be exploring how to use the new Solution Objects component of BlueFinity's mv.NET product to create a fully-formed, Visual Studio-aware business objects (BO) data access layer to your MultiValue database. By the term *business object* I mean an object- (or class-) based representation of the MultiValue database resident data, plus any additional additions or customizations of that data representation.

mv.NET is BlueFinity's flagship product. It is a .NET to MultiValue connectivity and productivity aid for developers wishing to create state of the art application user interfaces and web services for MultiValue-based applications.

The First Obvious Question

I guess one of the questions you might be asking at this point is "why would I want to create a business objects layer to my MultiValue database?" Well, if done properly, there are a number of very important benefits, some of which are detailed below:

1. A BO layer presents your MultiValue data in a generic, widely consumable manner. This allows many pieces of technology to consume and possibly update your MultiValue data. It also allows developers to focus on the task of creating line-of-business solutions and not on the database plumbing at the back end.
2. A BO layer provides an intuitive, strongly-typed data access mechanism. This allows more coding errors to be detected at design and coding time as opposed to at runtime. It also encourages developers to create more readable and maintainable code.
3. A BO layer allows the application developer to interact with the data persistence mechanism without the need to understand how that mechanism works. This means that the application developer does not need to understand MultiValue database technology to create MultiValue-linked applications.

Of course, other technologies allowing access to MultiValue data have been in existence for a number of years — ODBC connectors, ADO.NET providers, and XML data streams. However, it is only a properly formed, standards-based business objects layer that allows us to thoroughly address the challenge of providing flexible, high-performance, strongly typed, fully updateable database agnostic access to MultiValue data.

Reprinted with permission from the January/February 2010 issue of International Spectrum magazine.

The Solution Objects Toolset

No matter what approach you use, there are a number of discrete tasks that must be performed in order to end up with a meaningful BO access layer, some of the main ones being:

- Identify the entities that exist within your application;
- Identify which files hold data relating to these entities;
- Create a mapping between your entity properties and your database fields;
- Embellish the interface of your entity classes with additional functionality, such as the invoking of server-side routines, custom business logic, etc.;
- Define how relationships between entities are to be represented — for example: cross entity selection, lazy vs. eager data loading, cascaded deletes, and others;
- Define the exposure of entities and entity members to developers; and
- Manifest the above information and definitions into physical program code.

The purpose of Solution Objects is to dramatically reduce the complexity and timescale of performing these tasks.

The mv.NET Data Manager utility has been extended to include a range of entity modelling and code generation features. These new features are accessed using the new "Entity Models" node within the Data Manager's navigation treeview (fig. 1).

You are able to create any number of entity models. Typically you'll have one model per application. And, if required, you can create multiple versions of the same model — for example, one that's the current production version and one that's the current development version.

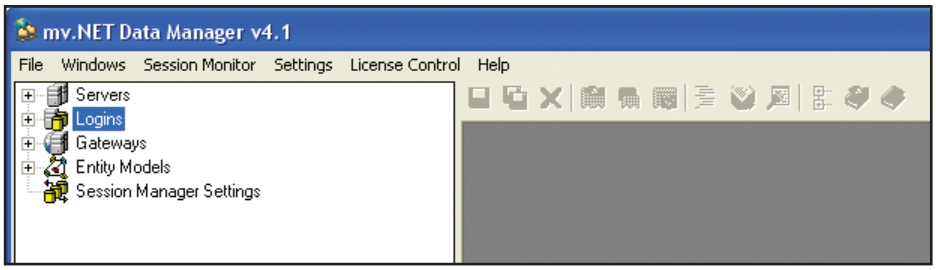

Fig. 1 The new "Entity Models" node within the mv.NET Data Manager

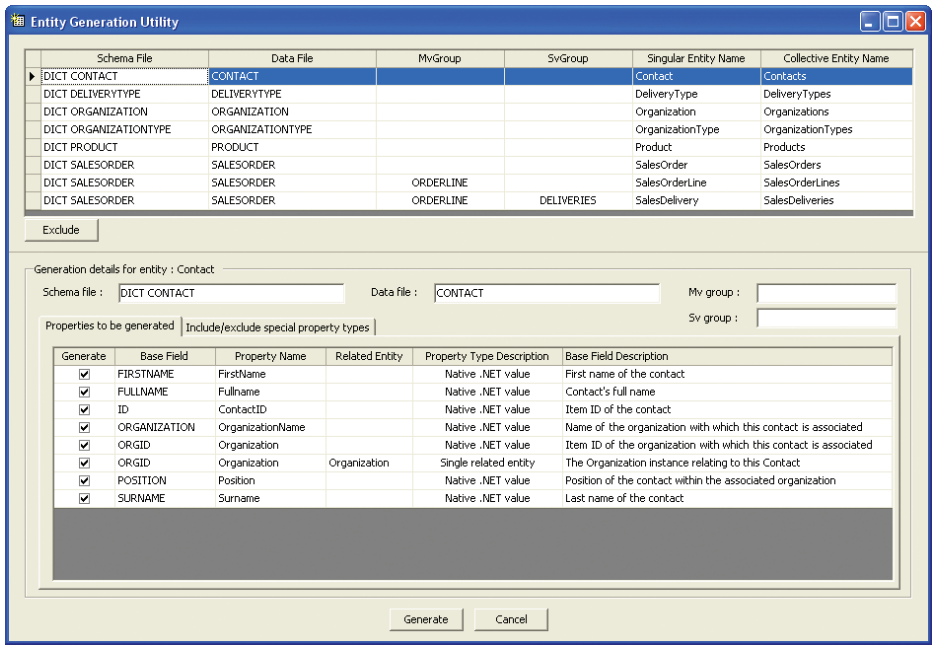

Fig. 2 The Entity Generator Utility Screen

What is an Entity?

An entity is a *thing* that your application deals with and, as such, can be a representation of pretty much anything — something physical or something abstract — it all depends on the *domain* that your application deals with.

If your application is a stock control system, your entities will probably be things like products, suppliers, purchase orders, etc. If you are creating a banking application, your entities are likely to be things such as bank accounts, customers, etc.

Defining Entities

The largest task that you will need to perform as part of your entity model definition process is the creation of entity definitions. However, the Data Manager provides an "Entity Generator" utility (fig. 2) which can kick-start this process by using your existing MultiValue dictionary items.

Data and Business Access Classes

Each entity within your model will have both a *Data Access Class* definition (DAC) and a *Business Access Class* definition (BAC). In fact, it can have multiple BACs.

The DAC is the definition of the class that will ultimately manifest the entity in hard program code — program code that the Data Manager will generate for you at the click of a button. As such, the DAC definition contains details of the properties and methods that are to be members of the class' interface. It is also the place where the mapping between the underlying database structure and the class content is defined.

The DAC definition allows you to specify "Selection Methods". These allow

Reprinted with permission from the January/February 2010 issue of International Spectrum magazine.

you to expose a specialized selection action as a method on the class's interface, allowing developers to use the full power of the MultiValue database selecting without needing to understand the *how*.

Finally, the DAC also allows you to specify "Subroutine Methods". These allow you to expose back-end MultiValue BASIC subroutines as methods on the class.

The Business Access Class definition is very different. Its main purpose is to allow you to control the exposure of DAC properties and methods to application developers — i.e., the developers who will be using the eventual BO layer. At the end of the day, it's the BACs with which the application interacts and references, not the DACs.

Business Access Layers

The final step before getting the Data Manager to generate some code for you is to define one or more Business Access Layers (BALs) within your model. The concept of a BAL is quite straight forward; it allows you to identify the BACs that are to be contained in a particular .NET assembly (dll) — this assembly being the file that you will distribute to the application developers to use as their BO access layer.

You can define any number of different BALs, thus allowing you to fine-tune the content of the various assemblies that manifest the model. However, you must define at least one BAL in order to generate code.

All of the above definitions (DACs, BACs, and BALs) are maintained using screens linked to the Data Manager's treeview (fig. 3).

Generating Code

The ultimate purpose of defining your entity model is to allow the Data Manager to generate source code to manifest the model as a series of .NET classes. This source code can then be included within a Visual Studio project in order to generate the final BO access assembly for distribution to your application developers.

The Code Generator allows you to define a number of things:

- the physical location of the generated code files,
- the language of the generated code (VB or C#), and
- the exposure of the constituent Data Access Classes, i.e., hidden or exposed.

Once the source code has been generated, you can include it within a Visual Studio project, build the project, and voilà — you have your BO access layer assembly. Job done!

The Fun Bit! Using the BO Layer

All that application developers need to do to use the BO access layer assembly is to add a reference to it within their Visual Studio application project. Once this is done, they can use it in a number of different ways. The first way is to use it programmatically.

For example, if we have a BAC called "Contact" with a property called "FirstName", we might have some VB code as shown in figure 4.

Let's look a bit more closely at the code.

The first line instantiates a DataRepository instance. This identifies to the BO assembly the place where the persisted data associated with the classes in the assembly can be found. In this case, we tell the assembly to use the mv.NET login profile called "SOP" (mv.NET login profiles contain database connection definitions).

The second line uses one of the CRUD methods automatically included within a BAC: "Read". Passed into the Read method are the DataRepository instance and the item ID of the required contact, "101". In our DAC, we defined the contact item ID to be a numeric value; therefore, the Read method forces an Integer data type to be supplied.

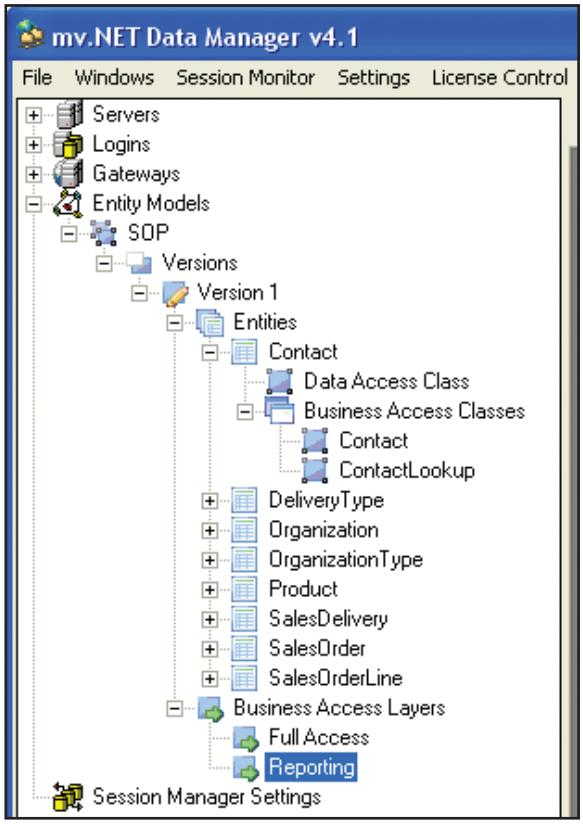

Fig. 3 Maintaining your Entity Model Definition using the Data Manager

```
Dim SOPdata As DataRepository = Repository.Initialize("Server=SOP")
Dim person As Contact = Contact.Read(SOPdata, 101)
Dim fName As String = Contact.FirstName
Contact.FirstName = "David"
Contact.Update
```

Fig. 4

Reprinted with permission from the January/February 2010 issue of International Spectrum magazine.

The third line extracts the value of the FirstName property.

The final two lines alter the FirstName property value and persist this data change to the database using another of the built-in CRUD methods: "Update".

Using Visual Studio Data Sources

A second way of using the BO assembly is to include it as a standard Visual Studio object data source. Once this is done, any component that can use an object data source will be able to access your data. One example of this is data binding. Both Web and WINForm applications support object data sources and the Code Generator automatically includes everything required to support bi-directional data binding.

Using Visual Studio's data sources window's "Add Datasource", you select an Object Data source and select a class from your BO assembly (fig. 5).

Once you've done this, the class will be listed in the Visual Studio Data Sources window and from there you can drag-and-drop onto the surface of your forms to your heart's content.

Summary

You can use a properly formed business objects access layer in lots of different ways. In doing so, it opens up your application development horizons and can be a great simplifier and time saver for application developers.

Watch the BlueFinity product space for further innovations over the coming months, including Silverlight integration – oops, that's done it, don't get me started on that one; you'll never get me to shut up! IS

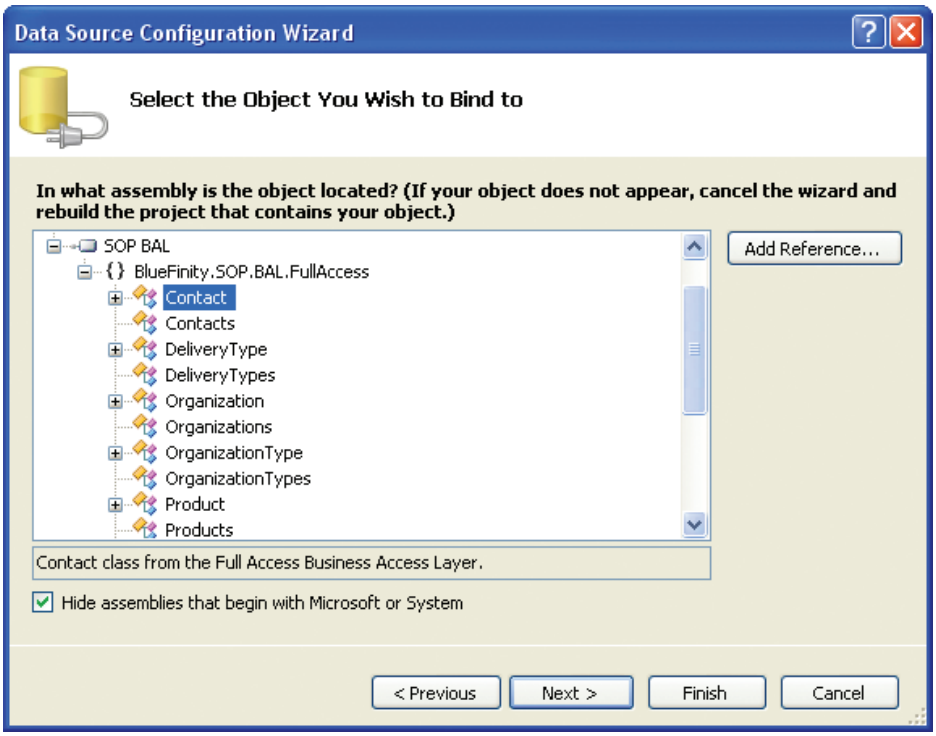

Fig. 5 Adding an Object Datasource in Visual Studio

Reprinted with permission from the January/February 2010 issue of International Spectrum magazine.