

Dali:

Can your DAL do this?

REVELATION TECHNOLOGIES

February 2, 2004

About Data Access Layers

One of the most daunting tasks in software development is getting the object oriented software to talk to the relational database. This is an area that many consider to be the least fun. Let's face it, software developers like writing software. They use object oriented techniques to write classes and algorithms. They think of the world in terms of objects and methods. Everything is an abstraction of this or an encapsulation of that. And objects are associated with one another through inheritance and aggregation. Anyone speaking a different language is irrelevant. On the other side of the universe are the database administrators and database analysts. To them, the world is one big database. In their world everything is represented as a row in a table. Things become associated with one another through relationships and foreign keys. Data integrity is maintained through constraints. And as the "keepers of the world" they are empowered and authorized to maintain the integrity of the world and defend it against those who would try to compromise it. Maybe I'm exaggerating, but not much. These really are two different worlds. Historically, getting these two worlds to communicate was never fun. Rather, it was viewed as a necessary evil.

At some point, someone came up with the concept of object oriented databases—databases that would store objects as objects rather than as rows of data. It was a good concept. It relieved the programmers from having to worry about key constraints, referential integrity and other things those relational databases were always complaining about. And since the database's "schema" was determined entirely from the application software that used it, there was no need for data modeling and all of those other non-essential things DBAs do. It truly was a programmer's paradise. But OO databases never really caught on. One reason was because programmers found that storing and retrieving objects was not quite as simple or intuitive as they hoped it would be. Many OO databases were also not as fast as their relational counterparts. But one of the biggest disadvantages of OO databases was the difficulty in querying them outside of a custom application. Remember that the data is stored as objects. So you can't do a simple query on the data like you could with a relational database. You can't even buy third-party reporting software like you could for relational databases. If you want to create reports you need to use whatever tools are provided by the database manufacturer or write your own. After about a decade, OO databases have now found their place in society. There are quite a few good uses for them, but seldom do these uses include storing large amounts of mainstream corporate data. This area is still dominated by relational databases.

Fortunately there are some software developers who can also speak relational. These are the folks to whom we give our gratitude, as well as large sums of money, for they are the ones who give us the *Data Access Layer*. A Data Access Layer or *DAL* is typically implemented as a collection of classes that represent database tables. These encapsulate all the necessary functionality for storing and loading data to and from the database. They provide an object oriented interface to application software while providing a database-specific interface to the database. Software developers who use a DAL are almost completely protected from the ugliness of the underlying database. The only developers who are exposed to this ugliness are the ones who write or maintain the DAL. If your DAL was written by someone else and requires no maintenance, then you truly are in a programmer's paradise.

About Dali

Dali is a self-configuring *intelligent* data access layer that connects your .NET classes to your relational database—without code generation and with little or no code changes. By analyzing your classes and database structure, Dali dynamically generates and caches all the necessary SQL to perform create, read, update and delete (CRUD) operations; automatically and behind the scenes. There is no need to know SQL or even ADO.NET. Dali abstracts these inside a simple class library. Easy to use methods like `Load()` and `Save()` do all the work of loading an object's state from the database and saving the object's state back to the database. If you know basic object oriented principles and can create .NET classes, you already have what it takes to use Dali. In some cases, you can have your existing code using Dali within minutes.

Dali is a versatile library with a lot of flexibility. Rather than lock you into a framework that you are not comfortable with, Dali allows you to choose your level of dependency and amount of integration code required. At its simplest, Dali can store objects that have absolutely no knowledge of it. This means there is no custom coding required to get your existing classes loaded from and saved to database tables. This level of freedom, however, comes with a few restrictions in what Dali can do for you. At the other end of the spectrum, you can integrate your classes tightly with Dali. This gives you a simpler programming model and far more features. This paper explains some of the features and capabilities of Dali. For a deeper understanding of what Dali can do, download the User's Guide.

No Code Generators

Typically, developers implement a data access layer by using a code generator that interrogates the database and creates a set of classes to represent the tables. Developers then have the option of using these classes as-is and implementing their business objects on top of this layer, or modifying the generated code. Each of these scenarios has a down side. In the former case, there is an extra layer of code. Business objects must interact with data objects. More code, more complexity, more things to break. In the latter case, developers modify the generated classes such that they can become the business objects. The ability to do this, however, relies on the code generator's ability to ignore the changes made by developers. If—or, more likely, *when*—the database schema changes these data objects will need to be regenerated. If the code generator is not able to accurately incorporate your changes you may end up rewriting your additions again. Many code generators allow you to make changes as long as you stay outside critical areas of the generated code, or enclose your changes inside special tags. This imposes unnecessary restrictions on developers and introduces more complexity. In either case, changes to the database require regeneration of code.

Dali does not rely on code generation. In most cases your business objects can access your database directly with little or no modification.

Data Mapping

The key to Dali's ease of use and automatic configuration is the way it exploits features of the Microsoft® .NET Framework. One of the features it relies on is *reflection*. Using the reflection capabilities in .NET, Dali can examine a class and discover all sorts of information such as the name of the class as well as names and types of fields and properties. Dali also uses the capabilities of ADO.NET to examine the database and determine names of tables and names and data types of columns. The first time Dali encounters one of your classes it interrogates it and gathers all necessary information. It then caches this data so the class does not need to be interrogated when it is encountered again.

In its simplest form, Dali can use these .NET features to load and store any ordinary .NET class to a database table. The class does not need to have any Dali-specific code in it. Dali does this by assuming that the name of the class is the same as the name of the table. It further assumes that the public fields and properties in the class have corresponding columns in the table with the same names. If you have the luxury of creating the class and the table, it can be this simple. In most cases, however, this is not true. Although a class may map directly to a table, the class name usually will not be the same as the table name, nor will the field and property names be the same as their corresponding column names.

This is handled with another feature of the .NET Framework: *Attributes*. The .NET Framework allows all classes, methods, fields and properties to have attributes associated with them. These attributes can be used to describe how the item should be used or to provide extra information about the item. With Dali, attributes are used to indicate how a class and its fields and properties map to the database. The attributes used by Dali are `DaliClassAttribute` and `DaliMemberAttribute`. These attributes can be associated with classes and members (fields and properties), respectively. The constructors for these attributes take parameters that indicate how the class, field or property maps to a table or column in a database. So, for example, to map your class to a table named `my_table`, you would simply add an attribute to your class definition:

```
[DaliClass (TableName="my_table" ) ]
public class MyClass { ... }
```

Dali now knows that data in class `MyClass` will be loaded from and saved to the table `my_table`. Likewise, if the field and property names do not match the column names you could add `DaliMemberAttribute` attributes to the fields and properties:

```
[DaliMember (ColumnName="cust_id" ) ]
public string CustomerID;
```

The final .NET feature exploited by Dali is *events*. Events are used to notify objects that something noteworthy has happened, thereby allowing the object to take appropriate action. If you derive your class from `DaliObject` or implement the `IDaliObject` interface, your class will raise events whenever a load, save or change occurs. These events can be captured to perform whatever actions you need to take. One common use is to load and save related objects. This is described in *Composite Objects*, below. There are pre- and post-events for every database operation. You may use these any way you wish.

Composite Objects

How would you handle composite (nested) objects? For instance, what if you have an `Order` object that contains a collection of `OrderDetail` objects. Suppose when you load the order you want all the details loaded with it. Likewise, when you save the order you want all the associated details saved. Most DALs don't address this. They leave it completely up to the developer to design a scheme for managing this. Dali provides some capabilities that make handling these situations a little easier. In a case like this, the first step is to decide whether you want the details stored as individual rows in a related table, or stored as a blob (binary large object) within the main object's row. To store it as a blob you would simply map the collection object to a column in the main table with a binary data type. For example, if an `Order` class has an `ArrayList` field that contains `OrderDetail` objects, simply having that field map to a binary column will automatically serialize the entire collection to a byte

array and write it to the database. Similarly, when the Order object is loaded, the entire OrderDetail collection will be deserialized into the ArrayList. If, on the other hand you prefer to store all the details in separate rows in a related table (the more common relational approach), you could easily do this using events. To do this, you would create event handlers for the Inserted and Updated events. These handlers would simply iterate the collection, saving each OrderDetail object they found. Loading the collection would be easy as well. The event handler for the Loaded event could simply use one of the Find methods to find and create all related OrderDetail objects and add them to the collection. Each of these handlers will only require a few lines of code.

Singletons

What do you do if there is a possibility that the application may ask for an object that was previously requested and is still in memory? Perhaps you have a multithreaded application (such as a web site) that allows multiple users access to the same objects. The first user loads an object and makes some changes. Before saving the object the second user loads the same object. Here, the term *same object* means having the same key value(s). Using a conventional data access layer (or using Dali in its default mode), the second user would get a completely new instance loaded from the database. This object would not reflect the pending changes made by the first user. The second user may make different changes than the first user. The user who saves last will “win.” Dali can easily be configured for *singleton* mode. Singleton mode means that if two or more users request the same object they all get a reference to the exact same instance of the object. Changes made by one user are immediately reflected to other users. That way there are no surprises when the object is saved. Depending on your needs, this is an option you may or may not want to exercise.

Summary

This paper only scratches the surface of Dali’s capabilities. Dali supports transactions and various isolation levels (if supported by the database) to ensure data integrity. Dali can be configured to require attributes or automatically infer table and column names. Dali can manipulate objects from multiple databases within the same application. So your data can come from two or more different databases—possibly even different types of databases (e.g., Microsoft® SQL Server™ and Microsoft Access).

Listing all of Dali’s features would be beyond the scope of this paper. But hopefully by now you have some understanding of what Dali can do. To get a deeper understanding of what Dali can do—and specifically, what it can do for you—download the trial version along with complete documentation and samples from Revelation Technologies at: www.revtechnologies.com.