Microsoft.NET Application Architecture: the Data Access Layer

11 July 2006

**Designing and building a robust data access layer**

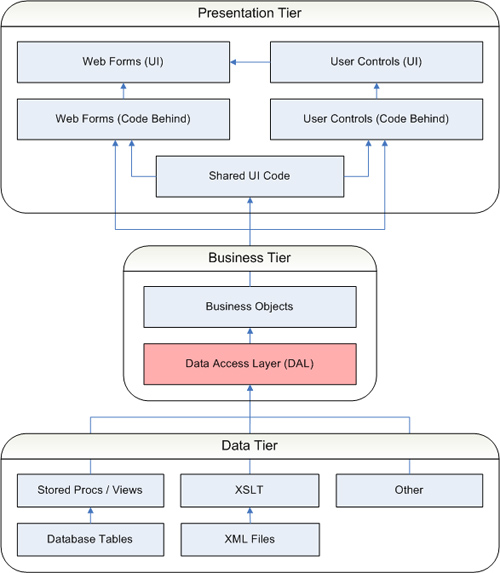
Building an understanding of architectural concepts is an essential aspect of managing your career. Technical interviews normally contain a battery of questions to gauge your architectural knowledge during the hiring process, and your architectural ability only becomes more important as you ascend through the ranks. So it's always a good idea to make sure you have a good grasp on the fundamentals. In this article you will explore a key component of application architecture known as the Data Access Layer (DAL), which helps separate data-access logic from your business objects. The article discusses the concepts behind the DAL, and the associated PDF file takes a look at a full-blown DAL implementation. This is the first in a series of articles discussing some of the cool things you can do with a DAL, so the code and concepts in this article form the base for future discussions.

**Layered design and the data access layer**

Layered application designs are extremely popular because they increase application performance, scalability, flexibility, code reuse, and have a myriad of other benefits that I could rattle off if I had all of the architectural buzzwords memorized. In the classic three tier design, applications break down into three major areas of functionality:

* The data layer manages the physical storage and retrieval of data
* The business layer maintains business rules and logic
* The presentation layer houses the user interface and related presentation code.

Inside each of these tiers there may also exist a series of sub-layers that provide an even more granular break up the functional areas of the application. Figure 1 outlines a basic three tired architecture in ASP.NET along with some of the sub-tiers that you may encounter:



*Figure 1 – Three tiered ASP.NET application with sub-tiers*

**The presentation tier**

In the presentation layer, the code-behind mechanism for ASP.NET pages and user controls is a prominent example of a layered design. The markup file defines the look and layout of the web form and the code behind file contains the presentation logic. It's a clean separation because both the markup and the code-behind layers house specific sets of functionality that benefit from being apart. Designers don't have to worry about messing up code to make user interface changes, and developers don't have to worry about sifting through the user-interface to update code.

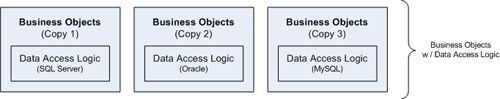
**The data tier**

You also see sub-layers in the data tier with database systems. Tables define the physical storage of data in a database, but stored procedures and views allow you to manipulate data as it goes into and out of those tables. Say, for example, you need to denormalize a table and therefore have to change its physical storage structure. If you access tables directly in the business layer, then you are forced to update your business tier to account for the changes to the table. If you use a layer of stored procedures and views to access the data, then you can expose the same logical structure by updating a view or stored procedure to account for the physical change without having to touch any code in your business layer. When used appropriately, a layered design can lessen the overall impact of changes to the application.

**The business tier**

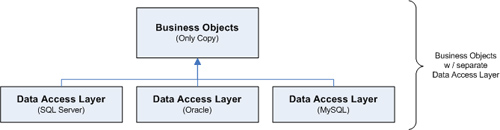
And of course, this brings us to the topic of business objects and the Data Access Layer (also known as the DAL), two sub-layers within the business tier. A business object is a component that encapsulates the data and business processing logic for a particular business entity. It is not, however, a persistent storage mechanism. Since business objects cannot store data indefinitely, the business tier relies on the data tier for long term data storage and retrieval. Thus, your business tier contains logic for retrieving persistent data from the data-tier and placing it into business objects and, conversely, logic that persists data from business objects into the data tier. This is called data access logic.

Some developers choose to put the data access logic for their business objects directly in the business objects themselves, tightly binding the two together. This may seem like a logical choice at first because from the business object perspective it seems to keep everything nicely packaged. You will begin noticing problems, however, if you ever need to support multiple databases, change databases, or even overhaul your current database significantly. Let's say, for example, that your boss comes to you and says that you will be moving your application's database from Oracle to SQL Server and that you have four months to do it. In the meantime, however, you have to continue supporting whatever business logic changes come up. Your only real option is to make a complete copy of the business object code so you can update the data access logic in it to support SQL Server. As business object changes arise, you have to make those changes to both the SQL Server code base and the Oracle code base. Not fun. Figure 2 depicts this scenario:



*Figure 2 – Business objects with embedded data access logic*

A more flexible option involves removing the data access logic from the business objects and placing it all in a separate assembly known as the DAL. This gives you a clean separation between your business objects and the data access logic used to populate those business objects. Presented with the same challenge of making the switch from Oracle to SQL Server, you can just make a copy of the Oracle DAL and then convert it to work with SQL Server. As new business requirements come in, you no longer need to make changes in multiple locations because you only maintain a single set of business objects. And when you are done writing the SQL Server DAL, your application has two functional data access layers. In other words, your application has the means to support two databases. Figure 3 depicts separating data access logic out into a separate DAL:



*Figure 3 – Business objects with separate data access layer*

**Design principals in the data access layer**

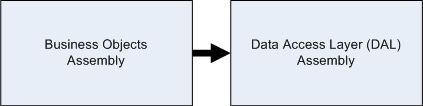
The objective of the DAL is to provide data to your business objects without using database specific code. You accomplish this by exposing a series of data access methods from the DAL that operate on data in the data-tier using database specific code but do not expose any database specific method parameters or return types to the business tier. Any time a business object needs to access the data tier, you use the method calls in the DAL instead of calling directly down to the data tier. This pushes database-specific code into the DAL and makes your business object database independent.

Now wait, you say, all you've accomplished is making the business objects dependent on the DAL. And since the DAL uses database-specific code, what's the benefit? The benefit is that the DAL resides in its own assembly and exposes database-independent method signatures. You can easily create another DAL with the same assembly name and an identical set of method signatures that supports a different database. Since the method signatures are the same, your code can interface with either one, effectively giving you two interchangeable assemblies. And since the assembly is a physical file referenced by your application and the assembly names are the same, interchanging the two is simply a matter of placing one or the other into your application's bin folder.

*Note: You can also implement a DAL without placing it in a separate assembly if you build it against a DAL interface definition, but we will leave that to another article.*

**Exchanging Data with the DAL**

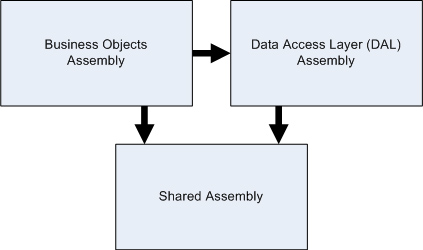
Now the question is: how do you exchange data between your business objects, the DAL, and vice versa? All interaction between your business objects and the DAL occurs by calling data access methods in the DAL from code in your business objects. As mentioned previously, the method parameters and return values in the DAL are all database independent to ensure your business objects are not bound to a particular database. This means that you need to exchange data between the two using non-database-specific .NET types and classes. At first glance it may seem like a good idea to pass your business objects directly into the DAL so they can be populated, but it's just not possible. The business object assembly references the DAL assembly, so the DAL assembly cannot reference the business object assembly or else you would get a circular reference error. As such, you cannot pass business objects down into the DAL because the DAL has no concept of your business objects. Figure 4 diagrams the situation:



*Figure 4 – Business objects assembly references the DAL, so the DAL has no concept of business objects*

**The custom class option**

One option is to pass information in custom classes, as long as those custom classes are defined in an assembly that both the business object and DAL assemblies can reference. From an academic standpoint, this approach is probably the truest form of a data abstraction for a DAL because you can make the shared classes completely data-source independent and not just database independent. Figure 5 depicts how the business object assembly and the DAL assembly can both reference a shared assembly:



*Figure 5 – The business object assembly and the DAL assembly both reference a shared assembly, so they can exchange information using classes and data structures from the shared assembly.*

In practice, I find that building out custom classes solely to exchange data doesn't give you much return for your effort, especially when there are other acceptable options already built into .NET.

**The XML approach**

You could opt to use XML since it's the poster child of flexibility and data-source independence and can easily represent any data imaginable. Of course, it also means that you will be doing a lot of XML parsing work to accommodate the data exchange, and I'm not a fan of extra work.

**The database interface approach**

You could also use the database interfaces from the **System.Data** namespace to exchange data between business objects and the DAL. Database specific objects such as **SqlDataReader**, **SqlCommand**, and **SqlParameter** are tied to SQL Server, and exposing them from the DAL would defeat the purpose. However, by exposing an **IDataReader**, **IDBCommand**, or **IDataParameter** object you do not tie yourself to particular database so they are an acceptable option, though not my first choice.

From an academic standpoint, the database interface objects do tie you to using a "database management system" even though they do not tie you to a specific database. Pure academics will tell you that the DAL should be "data-source independent" and not just "database independent" so be prepared for that fight if you have a Harvard or Oxford grad on your development team who majored in theoretical application design. Nobody else on the planet cares because the chances of your application moving away from a database system are fairly slim.

**My preferred approach: DataSets**

Another option for passing information, and the one that I gravitate towards because of its flexibility, is the DataSet. Microsoft created the **DataSet** class specifically for storing relational information in a non-database specific data structure, so the DataSet comes highly recommended for returning query information containing multiple records and or tables of data. Your work load shouldn't suffer too significantly from using the DataSet because DataAdapters, which fill DataSets with information, already exists for most database systems. Furthermore, getting data out of the DataSet is fairly easy because it contains methods for extracting your data as tables, rows, and columns.

Also note that a DataSet is technically data-source independent, not just database independent. You can write custom code to load XML files, CSV files, or any other data source into a **DataSet** object. Additionally, you can even manipulate and move information around inside the DataSet, something that is not possible with the database interfaces from the **System.Data** namespace.

**Exchanging non-relational data**

Of course, you also deal with non-relational information when you pass data back and forth between your business objects and the DAL. For example, if you want to save a single business object to the data-tier, you have to pass that business object's properties into the DAL. To do so, simply pass business object properties into the DAL via native .NET type method parameters. So a **string** property on your business object is passed into the DAL as a **string** parameter, and an **int** property on your business object is passed into the DAL as an **int** parameter. If the DAL updates the business object property, then you should mark the parameter with the **ref** modifier so the new value can be passed back to the business object. You can also use return values to return information as the result of a function when the need arises. Listing 1 contains examples of method signatures that you may need in the DAL if you have a **Person** business object in your application:

*Listing 1 – Data access layer method signature examples*

**DataSet** Person\_GetAll()

{

   //Returns a DataSet containing all people records in the database.

}

**DataSet** Person\_GetByPersonID(int personID)

{

   // Queries the database for the particular user identified by

   // personID. If the user is located then the DataSet contains a

   // single record corresponding to the requested user.  If the user

   // is not found then the DataSet does not contain any records.

}

**bool** Person\_Save(**ref int personID**, string fname, string lname, DateTime dob)

{

   // Locates the record for the given personID.  If the record exists,

   // the method updates the record.  If the record does not exist, the

   // method adds the record and sets the **personID** variable equal to

   // the identity value assigned to the new record. Then the method

   // returns the value to the business layer because **personID** is

   // marked with the **ref** modifier.

}

**int** Person\_DeleteInactive()

{

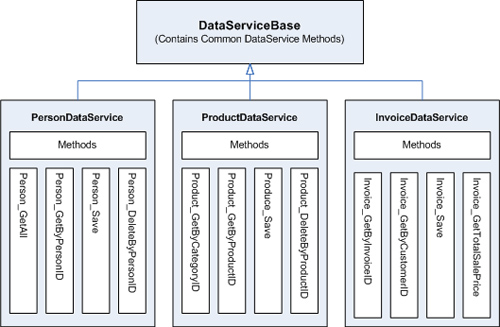
   //Deletes all inactive people in the database and returns a value

   //indicating how many records were deleted.

}

**Data service classes**

Normally you have one data access method in your DAL for each scenario in which you need to exchange data between a business object and the database. If, for example, you have a **Person** class then you may need data access methods like **Person\_GetAll**, **Person\_GetPersonByID**, **Person\_GetByLoginCredentials**, **Person\_Update**, **Person\_Delete**, and so on, so you can do everything you need to do with a **Person** object via the DAL. Since the total number of data access methods in your DAL can get fairly large fairly quickly, it helps to separate those methods out into smaller more manageable **Data Service Classes** (or partial classes in .NET 2.0) inside your DAL. Aside from being more manageable from a shear number standpoint, breaking down the DAL into multiple data service classes helps reduce check-out bottle necks with your source control if you have multiple developers needing to work on the DAL at the same time. Figure 6 depicts a DAL broken down into three individual data service classes:



*Figure 6 – Breaking down the DAL into multiple data service classes*

Notice that all of the data service classes depicted in Figure 3 derive from a single base class named **DataServiceBase**. The **DataServiceBase** class provides common data access functionality like opening a database connection, managing a transaction, setting up stored procedure parameters, executing commands, and so forth. In other words, the **DataServiceBase** class contains the general database code and provides you with a set of helper methods for use in the individual data service classes. The derived data service classes use the helper methods in the **DataServiceBase** for specific purposes, like executing a specific command or running a specific query.

**Putting theory into practice: the demo application**

At this point you should have a descent understanding of what the data access layer is and how it fits into an application from an architectural point of view. Theory is great, but at some point you have to quit talking and start coding. Of course, going from theory to practice is no trivial step, so I wanted to make sure you had a solid example to use as a foundation both in terms of code and understanding.

At the top of this article is a link to a zip file containing two items: a demo application containing a DAL implementation and a *Building a Data Access Layer* PDF that explains the code in detail. The application is fairly simple, a two page web app that allows you to view / delete a list of people on one page and to add / edit those people on another. However, it does implement all of the design principles that we've covered here. Enjoy!

**Test your implementation of Damon's  .NET Data Access Layer...**against realistic SQL Server data loads, using Red Gate's [SQL Data Generator](http://www.red-gate.com/products/SQL_Data_Generator/index.htm?utm_source=SimpleTalk&utm_medium=article&utm_content=DataGenNoteArmstrong&utm_campaign=sqldatagen).