

4/5/2009



MOBILECONTRIB

MOBILE CLIENT SOFTWARE FACTORY: DATA ACCESS APPLICATION BLOCK GETTING STARTED GUIDE

Contents

Introduction	3
Getting Started with the Data Access Application Block (DAAB).....	3
Understanding the Data Access Application Block Quickstart Application	3
Opening and Closing Databases.....	3
Creating a Database Object	3
Closing A Database.....	5
Executing T-SQL Commands	5
Creating and Dropping Tables.....	5
Executing Parameterised T-SQL Commands.....	6
Executing Commands that Return a Scalar	6
Retrieving Data	7
Fetching Records Using ExecuteReader.....	7
Working with SqlCeResultSets	7
Inserting Records	8
Updating Records.....	9
Retrieving Records	9
Performing Updates In a Transaction	10
Summary	11

Introduction

The purpose of the Data Access Application Block is to make writing code to access a SQL Server or SQL Server Compact Edition database less error-prone and more productive. It exposes methods to allow developers to execute T-SQL commands to modify data, or to return a `DbDataReader` or (in the case of a SQL Server Compact Edition target) a `SqlCEResultSet` to retrieve data.

This application block serves as a façade to ADO.NET, and requires that the developer is familiar with standard programming techniques with `DbCommand/SqlCeCommand` and `DbDataReader/SqlCeDataReader` objects. It simply provides easy to use wrappers around these objects that take care of correct disposal of objects and avoids the common programming errors that ADO.NET developers often make.

Getting Started with the Data Access Application Block (DAAB)

To use the DAAB, you need to include the following project from the `Mobile.codeplex.com` project in your application:

- `Mobile.DataAccess` – this contains the Data Access Application Block code

This block has no dependencies on any other blocks. Note that it does require a reference to `System.Data.SqlServerCe`. If the Quickstart does not compile straight away, it may be because you need to remove and re-add the reference to `System.Data.SqlServerCe` to match the version installed on your development PC.

Understanding the Data Access Application Block Quickstart Application

The Data Access Application Block Quickstart example application is very simple. It consists of a main form that displays a `DataGrid` control that is used to display a list of `Customer` objects the application retrieves from the database. When the user clicks on a row in the grid, a second form displays to allow the user to edit the fields of the `Customer` record shown in that row.

The application implements a very simple three tier architecture. All presentation logic is contained in the form, `Form1`. The business logic, which is very simple, resides in the `GetStartedBusinessLogic` class in the `Business` folder. All the data access code resides in the `DataLayer` class in the `Model` folder, which is where all application code that uses the DAAB is found. The `entities` folder contains the definition of the `Customer` class, which is an object that moves across different tiers of the application. Note that in a large application, one or more of these layers, and certainly the business entities, would reside in separate projects within your solution, but for simplicity in the Quickstart they all reside in the same project.

Opening and Closing Databases

Creating a Database Object

The first step in programming a data access layer using the DAAB is to create an instance of a `SqlDatabase` object, which takes a connection string as its parameter. The following code in

DataLayer.cs creates an empty database if one does not already exists and then creates a SQLiteDatabase instance, which is stored as a class field:

```
using Microsoft.Practices.Mobile.DataAccess;
using System.IO;
using System.Data.SqlServerCe;

public class DataLayer :IDisposable
{
    private SQLiteDatabase db;

    public DataLayer()
    {
        // Create database if it does not already exist
        string dbfilepath = GetApplicationDirectory() +
            @"\GettingStarted.sdf";
        string connectionString = "Data Source=" + dbfilepath;
        if (!File.Exists(dbfilepath))
        {
            SqlCeEngine eng = new SqlCeEngine(connectionString);
            eng.CreateDatabase();
        }

        // Get a connection to the database
        db = new SQLiteDatabase(connectionString);

        // The Database.GetConnection() method opens the shared
        // connection if it is not already open.
        // NOTE: Opening it here is optional - it will be opened the
        // first time you call any methods that use the database, but
        // doing it now may be preferable if you prefer to take the
        // performance hit now rather than later.
        db.GetConnection();

        // Note, you do not have to close this connection -
        // Database.Dispose() takes care of that
    }
}
```

Note that the DAAB creates a shared connection to the database but doesn't open it. The first time you open a connection to a SQL Server Compact Edition database, the SqlCE engine loads the database into shared memory, but subsequent connections to the same database simply access the already loaded database with very little performance hit. Therefore it is good practice to keep a 'phantom' connection open the entire time your app is running to ensure the database remains loaded.

As the comments in the code suggest, you can call the Database.GetConnection() method to open the shared connection if it is not already open, but if you do not do it here, it will just be opened the first time you call any DAAB methods that use the database; doing it at the time of initialisation may be preferable if you prefer to take the performance hit now rather than later.

Closing A Database

When your application has finished with a database, you must dispose of it to ensure all native resources are released. The best way of doing this is to make your own data access class implement `IDisposable`, and in the `Dispose` method, dispose of the `Database` instance:

```
public class DataLayer :IDisposable
{
    private SqlDatabase db;

    ...

    public void Dispose()
    {
        db.Dispose();
    }
}
```

In the Quickstart code, the `DataLayer` class instance is created and disposed of by the `Dispose` method of the `GetStartedBusinessLogic` class which also implements `IDisposable`; the `GetStartedBusinessLogic` instance is created by `Form1`, and disposed of by the `Form1_Closing` event handler.

Executing T-SQL Commands

The Quickstart application demonstrates a number of different ways of executing T-SQL commands. Some of these demonstrate different ways of achieving the same goal, for example using T-SQL or `SqlCeResultSet` to insert and modify records.

Creating and Dropping Tables

The `Initialize` method of the `DataLayer` class shows how to call the **`TableExists`** method and how to execute a simple, non-parameterized T-SQL command using the **`Database.ExecuteNonQuery`** method:

```
public void Initialize()
{
    // See if a table exists
    if (db.TableExists("Customers"))
    {
        // Execute a command to delete a table - use a text literal
        // for the command text
        db.ExecuteNonQuery("DROP TABLE Customers", null);
    }

    // Create a table - this time get the command text from resources
    db.ExecuteNonQuery(Properties.Resources.CreateTable, null);
}
```

Notice that the final call to `ExecuteNonQuery` retrieves the command text for the T-SQL `CREATE TABLE` command from resources. You will find the `CreateTable.sql` file containing the command in

the Model\SQL Scripts folder. If you create T-SQL command text in a file using Visual Studio, since it has the '.sql' extension, you benefit from intellisense support as you edit, and you can also test it against a real database during development. Once complete, you use the Add Existing File option in the resources editor to load it into resources.

Executing Parameterised T-SQL Commands

The commands described above did not take any parameters. The Quickstart example also shows how to execute a parameterised query. The InsertCustomer command is defined in a resource file as:

```
INSERT INTO Customers([Name], Title)
    Values (@Name, @Title)
```

To supply parameter values, use the **ExecuteNonQuery** overload that takes a **DbParameter[]**. Build a DbParameter array containing values for each of the named parameters ("@Name" and "@Title" in this example):

```
public int InsertCustomerTSQL(string name, string title)
{
    // First by a parameterized T-SQL command
    System.Data.Common.DbParameter[] parameters =
        new System.Data.Common.DbParameter[]
        {
            // Note that you do not need to precede the parameter
            // name with "@" - the DAAB
            // checks if it is there and adds it if necessary
            db.CreateParameter("Name", name),
            db.CreateParameter("Title", title)
        };

    // The InsertCustomer resource contains
    //"INSERT INTO Customers([Name], Title) Values (@Name, @Title)"
    db.ExecuteNonQuery(Properties.Resources.InsertCustomer,
        parameters);

    // Return the CustomerId value of the record just inserted
    return Convert.ToInt32(
        db.ExecuteScalar("SELECT @@IDENTITY", null));
}
```

Note: When you are inserting records, executing a T-SQL command does not yield the best performance. Best performance is achieved by using a SqlCeResultSet, as described later in this document.

Executing Commands that Return a Scalar

The previous code example also showed how to execute a command that returns a single value using the Database.ExecuteScalar method. If the T-SQL command you execute returns a row of data, then only the value in the first column will be returned.

```
// Return the CustomerId value of the record just inserted
return Convert.ToInt32(
```

```
db.ExecuteScalar("SELECT @@IDENTITY", null);
```

This statement returns the value of the database assigned identity field of the Customer table record – in other words the value of the CustomerId field of the record that was just inserted.

Retrieving Data

When you are retrieving data, you have two choices. If you need to perform a complex SELECT statement involving joins between two or more tables, then you must use the **ExecuteReader** method and a suitable T-SQL command string. If you are retrieving records from a single table, then you will get best performance by using a *DbDataReader* or a *SqlCeResultSet* in Table Direct mode. See *Working with SqlCeResultSets – Retrieving Records* later in this document for an explanation of how to do the latter.

Fetching Records Using ExecuteReader

Although this is not demonstrated by the Quickstart example application, the technique is easily understood as it follows the pattern already described for executing other T-SQL commands.

For example, to execute a T-SQL SELECT command, use code such as the following:

```
string cmdText = "SELECT * FROM Customers " +
    "JOIN Orders ON Orders.CustId = Customers.CustomerId"
using (DbDataReader rdr = db.ExecuteReader(cmdText, null))
{
    // Do something ...
}
```

You can use a parameterised query and pass parameter values using a *DbParameter[]* in the same way as described previously.

IMPORTANT: Notice the use of the `using {...}` statement above, used to close and dispose of the data reader. You must dispose of a *DbDataReader* instance when you have finished using it.

Working with SqlCeResultSets

When working with SQL Server Compact Edition databases, best performance is achieved by using a *SqlCeResultSet*, which extends a *DbDataReader* to allow insert and update capabilities.

You cannot insert or update data in more than one table at a time using a T-SQL command. Therefore for insert and update operations you can get best performance by using a *SqlCeResultSet* in Table Direct mode, which bypasses the query processor component of the SQL Server Compact Edition engine and allows you to work directly against the base table. You will also get better record retrieval performance if you are only selecting records from one table.

When you use a *SqlCeResultSet* you have to write a little more code than if you use the *ExecuteNonQuery*, *ExecuteScalar* and *ExecuteReader* methods already discussed. You must create your own *SqlCeCommand* instance and configure it for Table Direct access, and then call the

ExecuteResultSet method which returns a SqlCeResultSet. As with ExecuteReader, you must be sure to dispose of the SqlCeResultSet when you have finished with it. The basic pattern looks like this:

```
// First we must create a new command object and set it to TableDirect
using (SqlCeCommand cmd =
    new SqlCeCommand("Customers", (SqlCeConnection)db.GetConnection()))
{
    cmd.CommandType = System.Data.CommandType.TableDirect;

    using (SqlCeResultSet rsltSet = db.ExecuteResultSet(cmd,
        ResultSetOptions.Scrollable | ResultSetOptions.Updatable))
    {
        // Do something ...
    }
}
```

Inserting Records

To insert records using a SqlCeResultSet, call **Database.ExecuteResultSet** in the way just described to get a SqlCeResultSet, call **SqlCeResultSet.CreateRecord** to return a **SqlCeUpdatableRecord** instance, set the field values in the SqlCeUpdatableRecord instance and insert it into the SqlCeResultSet. The **InsertCustomers** method in the sample application does this:

```
public void InsertCustomers(List<Customer> customers)
{
    // The fastest insert technique is to use SqlCeResultSet
    // First create a new command object and set it to TableDirect
    using (SqlCeCommand cmd = new SqlCeCommand("Customers",
        (SqlCeConnection)db.GetConnection()))
    {
        cmd.CommandType = System.Data.CommandType.TableDirect;
        using (SqlCeResultSet rsltSet = db.ExecuteResultSet(cmd,
            ResultSetOptions.Scrollable | ResultSetOptions.Updatable))
        {
            rsltSet.Read(); // Position basetable cursor

            foreach (var customer in customers)
            {
                SqlCeUpdatableRecord newrec =
                    rsltSet.CreateRecord();
                newrec.SetString(
                    newrec.GetOrdinal(Customer.NAME_COLUMN),
                    customer.Name);
                if (customer.Title != null)
                {
                    newrec.SetString(
                        newrec.GetOrdinal(Customer.TITLE_COLUMN),
                        customer.Title);
                }
                else
                {

```



```

        newrec.SetValue(
            newrec.GetOrdinal(Customer.TITLE_COLUMN),
            DBNull.Value);
    }

    // Add the record
    rsltSet.Insert(newrec);
}
}
}
}
}

```

Updating Records

Modifying a record using a `SqlCeResultSet` is achieved in a very similar way. See the **SaveCustomer** method in the QuickStart project for an example.

Retrieving Records

To retrieve records, you get a `SqlCeResultSet` instance in the same way and read through the records to build an array of objects to return to the caller. The sample application uses LINQ to build an array of customer objects:

```

public List<Customer> GetCustomers()
{
    List<Customer> customers = null;
    // The fastest retrieval technique is to use a SqlCeResultSet
    // First create a new command object and set it to TableDirect
    using (SqlCeCommand cmd = new SqlCeCommand("Customers",
        (SqlCeConnection)db.GetConnection()))
    {
        cmd.CommandType = System.Data.CommandType.TableDirect;
        using (SqlCeResultSet rsltSet = db.ExecuteResultSet(cmd,
            ResultSetOptions.Scrollable | ResultSetOptions.Updatable))
        {
            // Select each record and build a new Customer object
            // for each
            var custQuery = from SqlCeUpdatableRecord c in rsltSet
                            select new Customer()
            {
                CustomerId =
                    c.GetInt32(c.GetOrdinal(Customer.CUSTOMERID_COLUMN)),
                Name =
                    c.GetString(c.GetOrdinal(Customer.NAME_COLUMN)),
                Title =
                    c.IsDBNull(c.GetOrdinal(Customer.TITLE_COLUMN))
                    ? null :
                    c.GetString(c.GetOrdinal(Customer.TITLE_COLUMN))
            };
            // Convert to a List
            customers = custQuery.ToList();
        }
    }
}

```

```

        return customers;
    }

```

Performing Updates In a Transaction

If you want to perform a series of updates in a transaction you call the `Database.GetConnection()` method to get the `SqlCeConnection` object, then call `BeginTransaction()` to get a `DbTransaction` instance. You must again create a `SqlCeCommand` object and you can set that to be associated with the `DbTransaction` instance, perform your updates, then call `DbTransaction.Commit`, or `.Rollback`.

You can do this with any of the Database methods described in this document, **ExecuteNonQuery**, **ExecuteScalar** or **ExecuteResultSet**. The sample application demonstrates how to do this using **ExecuteNonQuery**:

```

public void InsertCustomersInTransaction(List<Customer> customers)
{
    // If you want to use a Transaction, do it like this:
    using (System.Data.Common.DbTransaction txn =
        db.GetConnection().BeginTransaction())
    {
        try
        {
            // We can use any of the techniques shown above, but we
            // must create our own SqlCeCommand object, so we can
            // associate it with the transaction
            using (SqlCeCommand cmd = new SqlCeCommand(
                Properties.Resources.InsertCustomer,
                (SqlCeConnection)db.GetConnection(),
                (SqlCeTransaction)txn))
            {
                // Add the parameters to the command ourselves,
                // instead of getting the DAAB to do it
                // - that way we can reuse the command rather than
                // creating a new one for every insert
                cmd.Parameters.Add(db.CreateParameter("Name",
                    System.Data.DbType.String, 50, string.Empty));
                cmd.Parameters.Add(db.CreateParameter("Title",
                    System.Data.DbType.String, 20, string.Empty));

                foreach (var customer in customers)
                {
                    // Reuse the command with different parameter
                    // values
                    cmd.Parameters[0].Value = customer.Name;
                    if (customer.Title != null)
                    {
                        cmd.Parameters[1].Value = customer.Title;
                    }
                    else
                    {
                        cmd.Parameters[1].Value = DBNull.Value;
                    }
                }
            }
        }
        catch { }
    }
}

```

```

        }
        db.ExecuteNonQuery(cmd, null);
    }

    // Commit the transaction
    txn.Commit();
}
}
catch (SqlCeException exc)
{
    // Rollback the transaction
    txn.Rollback();
    // rethrow
    throw new ApplicationException(
        "Error inserting records through transaction", exc);
}
}
}

```

Summary

The Data Access Application Block is a thin wrapper around standard ADO.NET usage that makes database programming less error prone and therefore more productive. Using it, you can write less code to build your own data access layer and are less likely to make errors such as forgetting to dispose of ADO.NET data objects.