Visual Studio and MySQL

Chapter 15 introduced the basics of using MySQL with .NET and Visual Studio. In this chapter we find out whether you can stay with MySQL for substantial Visual Studio database application development requiring a lot of coded application-database interplay, or whether such Visual Studio projects need to be migrated to SQL Server.

There's lots of code here. Feel free to lift it whole or in chunks, which include:

- how to get the info you need from `information_schema`,
- login forms, including reading from and writing to the Windows registry,
- populating ComboBoxes from a database,
- how to implement a clientside cache for database data,
- configuring DataGridViews to use such caches for tables of any size,
- databinding and update management with BindingSource objects,
- context menus,
- custom events,
- runtime query generation,
- on-the-fly lookup data lookup browse windows using a DataGridView,
- on-the fly lookup data lookup browse windows using a ListView,
- on-the-fly column *Find* dialogues,
- on-the-fly text editing for large text columns,
- how to write for multiple DBMS backends.

We assume installations of .NET 2.0 or later, Visual Studio 2005 or later, and MySQL 5.0.22 or later. To compare MySQL and SQL Server versions, you also need SQL Server or SQL Express.

Connector/NET 5/6

Connector/NET is at http://dev.mysql.com/downloads/connector/net/. Pick a stable release for your version of MySQL.

Before installing, uninstall any previously installed version. Installation of versions 5&6 is point-and-click.

If you are using a Connector/NET version before 6, once it is installed, navigate to its *docs* folder and drag a shortcut for *MySql.Data.chm* to the desktop, or wherever you keep your help files. To ensure that the Connector has installed itself in your Windows *Global Assembly Cache*, ensure that gacutil finds one MySQL.Data entry. ...

```
"%programfiles%\Microsoft Visual Studio 8\SDK\v2.0\Bin\gacutil.exe" /l MySQL.Data
```

Connector/NET is not a graphical design tool. For that you need MySQL for Visual Studio, first released in September 2006 and available in a separate package before the first release of Connector/NET 5.1, then bundled with Connector/NET, and now available as a separate product *here*.

We wrote *TheUsual* in PHP (*Chapter 12*) as a paging database browser for any data table or master-detail pair of them, in any MySQL database for which the user has appropriate privileges. It pages through any table; it supports row edits, inserts and deletes; and it can find a row on the master grid primary key. Can we write a more powerful version of this for MySQL Visual Studio 2005 or later?

- page the detail grid too, so it can also handle huge tables;
- give the application automatic popup lookup browse windows for any column that is a foreign key referencing another table;
- provide a find dialog in each grid for finding any column value;
- tooltips;
- make the app DBMS-agnostic—encapsulate MySQL-specific code so the app can be taught to run against another DBMS simply by replacing MySQL modules with modules for another DBMS (for example Sql Server 2005).

The specification

Login should connect to a database server via on-the-fly authentication of username and password, or optionally via user-selectable persistent logins, and it must leave behind a connection object for use by the application. SQL Server 2005 can use Windows authentication but MySQL cannot, so the login module must be DBMS-specific. We are writing for developers and DBAs rather than end-users, so we delegate database, table and column privileges to login control. The connection object needs a DBMS-agnostic wrapper.

Databases: Like *TheUsual* for PHP, the application should offer the user a list of available databases as reported by `information_schema`. Validation consists simply of setting the connection's default database to the user's choice.

Master table: Once a database is selected from a pick list, the application should offer a pick list of available tables or Views as reported by a query to `information_schema`.

Detail table: Once a master table is chosen from a pick list, the application should offer a list of detail tables having a foreign key that refers to the selected master table (another information_schema query). If a detail table is selected, the application should open coordinated master and detail grids for the two tables. Absent choice of a detail table, the application should simply browse the master table.

Grids should be paged if the rowcount justifies paging, should be quick regardless of table size, should offer clickable column sorting and searching.

Add, edit, delete, lookup: If a table has a primary key, browsing should permit adding new rows, adding rows seeded as copies of existing rows except for the primary key, deleting rows, and looking up foreign key values in popup browse windows. Popup lookup browses should be brisk even with large tables. They should support sorting on any column, and one-click selection. All data updates should be two-stage: (i) make the changes in the browser, (ii) commit them to the database. It must be possible to either undo stage (i) changes, or refresh the browser without saving them.

Flexibility: The application should permit the user to iteratively browse any available table, or table pair, simply by selecting databases and tables from pick lists.

Scalability: All this should work for any available table, large or small. Even for very large tables, paging or caching should provide crisp performance.

Language: C# is OO-friendlier than Visual Basic, and much easier to write and maintain than C++, so that's the choice.

The tools we need

The login dialog can be a standard Windows Form with TextBoxes for server name, username and password, and a CheckBox to tell *theUsual* whether to remember connection parameters. *TheUsual* itself can also can live in a standard Form. It doesn't need a menu (yet). Standard ComboBoxes are ideal for database and table selections. Ordinary Buttons will do fine for navigation, Go, Update and Exit interfaces. Visual Studio has a nice array of ToolStrip tools we can use for paging parameters. All these widgets can be dropped onto a form from the Visual Studio Toolbox, as is. It takes just a few minutes to name and arrange them as desired.

In early versions of Visual Studio, the browsing grid of choice was the DataGrid. VS 2005 and later sport the DataGridView, with enhancements that are especially useful for browsing database tables:

- data binding can be generalised through use of a BindingSource, hiding the details and simplifying code for queries and for updates;
- in VirtualMode, the DataGridView can cache very large datasources such that accessing any part of the table is nearly instantaneous; implementing VirtualMode is not trivial, but not forbidding either;

Unfortunately a DataGridView can turn just one of these tricks at a time. A BindingSource streamlines update code very nicely, but caching has to be done serverside, for example via a LIMIT clause in the query. With VirtualMode, on the other hand, you lose update streamlining, but automatic clientside caching lets you forget all about paging.

Since the main browsing grids must support updates, they should use a BindingSource with serverside caching via a LIMIT clause. Lookup grids can be readonly, so they can run in VirtualMode with a clientside cache. We will also write a ListView-based lookup grid.

The project thus needs only standard elements of VS 2005 or later, MySQL 5.0 or later and Connector/NET 5 or later. The SQL Server version of course needs an installation of that DBMS.

Class layout

Let the application namespace be `theUsual`, and let the principal class be a Windows Form subclassed as `TheUsual` with dropdown user input controls for selecting the database, master table and detail table; two data browsing grids; command buttons for filling the grids, for updating after edits, and for exiting from the program; a statusbar for general program messages; navigation buttons; and a toolstrip for paging parameters.

How to partition DBMS-agnostic and DBMS-specific code? DotNET has a set of DBMS-agnostic data classes (for example `DataTable`, `DataSet`, `BindingSource`) that mediate between DBMS-specific data classes (for example `MySqlConnection`, `MySqlCommand`, `MySqlDataAdapter`) and Windows controls (for example `ComboBox`, `DataGridView`). We will give *TheUsual* a thick DBMS-agnostic layer that talks to data-mediating classes, and a thin DBMS-specific layer.

How to implement that partition? One way would be to subclass all DBMS-specific functionalities. That will be useful here and there, but as a global approach it risks a blizzard of vexing cross-class references. The .NET concept of *partial class* offers a simpler overall solution. `Partial class foo` can exist in multiple files if each file declares it `Partial` in the same namespace. DotNET uses partial classes to separate code which it generates from code which you write. *TheUsual* will partition DBMS-specific and DBMS-agnostic code in the same way.

The specification also implies

- a DBMS-specific login class;
- a manager class to encapsulate some DBMS-specific grid management;
- an info class for passing database and table info to a lookup browser;
- for lookups, a class that implements a data-driven DataGridView running in VirtualMode; we can write that class generically but it will need a custom database-specific subclass, too;
- a class implementing a lookup as a ListView, so we can compare its capabilities and performance to the VirtualMode DataGridView, and
- a class to implement column find.

Some files will be common to each DBMS implementation. Does Visual Studio support sharing common files across multiple projects? Sure, if you purchase Microsoft Visual Source Safe (VSS). But that's another $500. We do not assume you have a copy. Without VSS, files are shared across projects by copying them. Ugh. Are there Visual Studio tools for comparing source files? Sure, if you purchase the VS Team System for another $500. Without it, we can use familiar text editors (e.g., *TextPad*) for that task.

The *custom classes* are :

```
namespace theUsual {
  class TheUsual {                 •  TheUsual
    class DgvManager               •  manage DBMS specifics in grids
    class ListViewTheUsual         •  popup lookup ListView browse window
    class rowInfo                  •  row info passed to called classes
    class TheUsualConnection       •  connection object hiding DBMS specifics
  }
  class Cache                      •  clientside cache for class DgvFormJIT
  class DataRetriever              •  DBMS-specific data retriever for Cache
  class DgvFormJIT                 •  Just-in-time cached lookup browse window
  class dgvInfo                    •  grid info for called classes
  class DgvRowFinder               •  popup find dialogue
  class FindDlg                    •  dialog for searching for a column value
  class LoginForm                  •  DBMS-specific login dialog
  class ServerDlg                  •  dialog for choosing saved login
  class TextEditor                 •  popup text editor
}
```

To read the rest of this and other chapters, *buy a copy of the book*

TOC ***Previous*** ***Next***