

Introduction To Client-Server Using VFP and MySQL

Session Number 12

*Whil Hentzen
Hentzenwerke
980 East Circle Drive
Milwaukee WI 53217 USA
Voice: 414.332.9876
Email: whil@hentzenwerke.com*

In 1994, I reviewed Robert Green's Client-Server Applications with VFP book for the second Pros Talk Fox series. Thirteen years later, one would think that everyone and their brother (and their sister) has written a client/server application. Au contraire! While writing MySQL Client/Server Applications with Visual FoxPro this year, I found that there are large numbers of

folks who have never touched any client/server tools, but are, for one reason or another, now anxious to do so. So join the crowd!

In this session, we'll build a basic client/server application using VFP and the open source database engine MySQL. The pair make a great combination, powerful and hugely scalable, with nary a client-access license or fee in sight. We'll start out discussing the concepts that differentiate a native-DBF application with a client-server system. Then we'll look at the interactive use of VFP in connecting to a back-end and discuss the plumbing involved. Finally, we'll build several typical forms and show how they differ from the typical user interfaces you may be used to building in DBF-style systems.

A Client-Server State of Mind

The way we were – thank you, Richard Grossman

One of the major influences in building FoxPro applications was Richard Grossman's landmark demonstration application, Pro-Demo, that shipped with FoxPro 1.0 in the early '90s. It featured a file-card interface, where you could select a table and then navigate through every record in it using <Next>, <Previous>, <First>, and <Last> buttons. In the DOS world that FoxPro 1.0 lived in, that's what those buttons looked like, too – text strings surrounded by less-than and greater-than signs. In Windows, the toolbar looks a bit nicer, but the functions remain the same. See **Figure 1**. For more than a decade, Fox developers relied on that paradigm to design their user interfaces. Times have changed, but the interface has not.

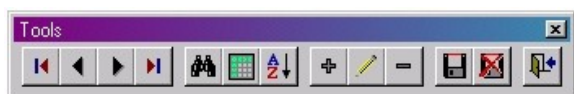


Figure 1. *The classic VCR navigation button toolbar.*

In some ways, that interface is timeless – witness the page navigation buttons in Adobe Reader or the playback buttons on your DVD player or iPod. What is different, though, is the idea that you can move through the entire table. Back in 1992, when a table frequently had 500, 1,000, or even 5,000 records, that interface was workable. In situations where the table held 10,000, 50,000, or... in some cases, 100,000 records, that interface was still workable, as long as you also had a <Search> button to help the user avoid having to click <Next> 74,200 times to get to “Slawinski.”

When the table has 37,000,000 records and is stored on a database server, you're just not going to want to – or be able to – have at the entire table in one fell swoop. And here is the crux of a client-server system – your first move is *always* to select the subset of records you want to look at, in order to narrow down that 37,000,000 to a more manageable number, like, say, 105.

Once you do so, many of the problems stay the same – moving from one to the next, changing the order of the records you're looking at, saving changes, and so on. To be sure, there are still some other differences. For example, while you're working with just a small subset of parent records, you may be also dealing with the entire set of values in a lookup table, which can introduce some new dynamics. But the fundamental difference between the single user and multi-

user LAN based systems you've been working with in the past is that you're now dealing with a small subset of the entire database.

Lest you get despondent, though, in bleak anticipation of having to relearn everything in order to build client-server apps, let me reassure you – one of the reasons Visual FoxPro is such an excellent client-server front end is once you bring down a record set from the back end, you can work with it using the VFP data handling commands you already know and love.

This makes VFP much easier to use than those other client-server front ends that have no concept of a 'record pointer', and force you to move through a small set of records with arcane syntax and workaround constructs. VFP was built with this type of work in mind!

The way we will be – you always get just a subset of records

When a user opens a table-centric form in a traditional application, the form loads up the data needed to populate the form, perhaps using the data environment of the form, and the first record is displayed in the form. The user would use the toolbar to navigate between records and then manipulate the record's data, delete the record, or add a new record.

With a client-server application, however, the form isn't simply opened with data available in it. Instead, the user needs to select one or more records, or (in nicely designed systems) add a brand new record from scratch from the very beginning. This is done via a filter form, which may be a separate form or one page in a page frame on a catch-all form. See **Figure 2** for an example of a filter on a separate form.

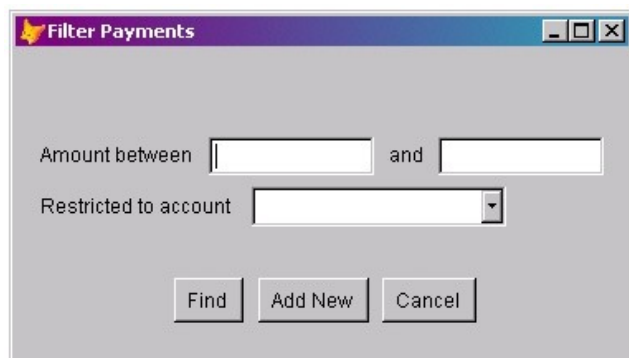


Figure 2. A typical form used to filter records in a client-server application.

Multi-user questions

One of the first questions people ask about client-server is "Is a client-server application inherently multi-user?" The answer is that this question involves two separate issues – the physical handling of the database files and the logical potential for collisions when two users are trying to update the same piece of data. The way a client-server application handles the first issue is different than how a traditional application handles it, but they both need to deal with the second issue the same way.

When a traditional application accesses a DBF file, either for reading or writing, it's actually touching the file itself via the operating system. As a result, the programmer has to worry about file locks (such as when reindexing) and record locks (when updating a record). With a client-server database back end, the database engine takes care of actually touching (reading and writing) the database. The programmer doesn't have to.

That said, the programmer does have to consider data integrity issues – what happens when two people are trying to write to the same data element? Let's look at two examples.

Suppose two users are trying to update the address field of a customer. In many traditional applications, this would be done via either GATHER MEMVAR or TABLEUPDATE(). In both cases, the entire row has to be locked in order for the one field to be changed. With a client-server application, a SQL UPDATE command is used instead, and the database engine handles the locking necessary.

The question remains: who wins? If Alvin is trying to change the address to "100 Albuquerque" and Bob is trying to change it to "200 Buffalo", whoever is last to touch the database is the one whose change sticks. While this could be a problem, it's more likely a problem in the business process – why are two different addresses being entered for the same record? Either one of the source documents is wrong, or the entity being updated is mistaken ("Oh, you mean there are *two* muffler/donut shops with the same name?").

On the other hand, there are legitimate situations of two users trying to update the same field at the same time. For example, one scenario involves the quantity field of an inventory record. Suppose two users are trying to decrement the quantity of the same item as they're placing orders for the item. In a traditional application, the first user locks the record, updates the field, and releases the lock. The second user has to wait for the first user's lock to be released before being allowed to issue their own update.

There are several ways to approach this issue with a database back end. The first is to use MySQL's record locking capability, similar to VFP. The second is to wrap the operation in a transaction. And the third is to use a semaphore field to indicate that the record is being edited; releasing the semaphore once the edit has been committed.

Development and Deployment Scenarios

The configuration of simple Visual FoxPro applications is fairly easy to understand – an EXE, some run-time files, and a few DBFs. But client-server applications using VFP and MySQL are more complex – there are more components, and there are more places for them to go. And the development and deployment environments are different as well.

For each environment, let's look at the pieces we've got, and then where they can go.

Development Scenarios – Components

There are four basic components you'll use during development. Visual FoxPro, of course, is the first. The ODBC Driver is the glue that connects VFP to MySQL. The MySQL database server, which is simply the engine that manages the data. And, finally, the MySQL GUI Tools, which while not required, can make your life a whole lot easier.

Visual FoxPro IDE

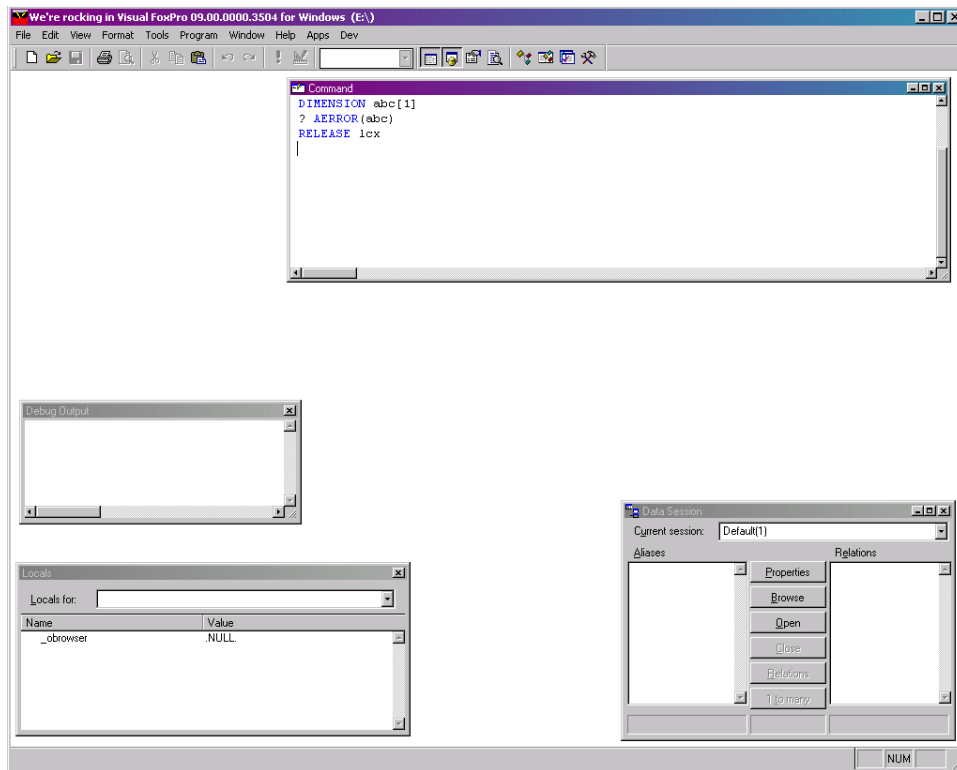


Figure 3. The Visual FoxPro IDE with the Command Window, Data Session, and two debugging windows.

ODBC Driver

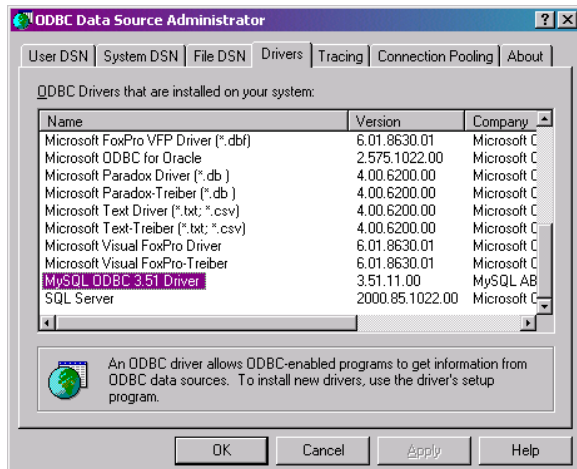


Figure 4. The ODBC Drivers page with the MySQL ODBC driver highlighted.

MySQL Engine

It's difficult to take a picture of a service running. It's sort of like taking a picture of a strong wind. But we'll try.

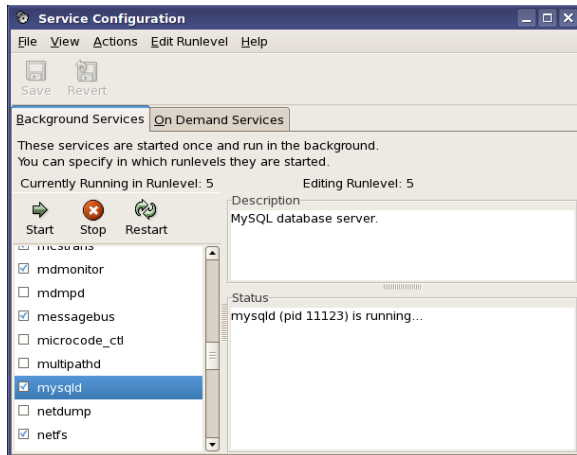


Figure 5. The MySQL Server service, *mysqld*, running on a Fedora Core Linux server.

MySQL GUI Tools

There are a number of GUI tools that work with MySQL. Two of these, the MySQL Administrator and the MySQL Query Browser, are produced by the MySQL folks, and should be an integral part of your toolkit. There are versions for Windows, Linux and Mac. More on these later.

Development Scenarios – Locations

In brief, for your development configuration, you:

1. Install Visual FoxPro on a Windows box,
2. Install the MySQL ODBC driver on the same Windows box,
3. Install the MySQL engine either on the same Windows box or a separate box (running Windows, Linux, Macintosh, or another OS), and
4. During installation, the database is installed on the same box as the MySQL engine.

If the MySQL engine is on a separate box from VFP, all that is required is for that separate box to be reachable by the Windows box via a TCP/IP connection.

Now that the pieces are in place, you connect to MySQL from VFP through the ODBC driver. Once connected, you can use remote views, SQL pass-through, or cursor adapters to bring selected data into your application, work with that data locally for a while, and then stuff it back into the database.

Development box: VFP/ODBC, GUI Tools

All of the action, from your point of view, is going to occur on your development box. Thus, you'll want VFP and the GUI tools for connecting to MySQL to be running on your workstation. You'll also need to have ODBC, which is a Windows technology, installed on your workstation.

Development box or Remote box: MySQL Server

Should you have MySQL installed on your workstation, or should you have it on a remote machine? You can argue both ways pretty successfully. Having it on your local machine means that you've got a self-contained environment; you can pick up and go, having everything you need right at hand.

On the other hand, having your MySQL Server installed on a separate machine means that if your workstation becomes unavailable (this is a Windows machine, after all), you can simply move over to another machine and connect from there. This will also give you practice and experience in connecting to a remote MySQL Server. You don't want to be one of those developers who does everything locally, until the night before deployment, only to find that they are missing some critical concept of remote connectivity.

Deployment Scenarios – Components

When you deploy, the components you'll use differ somewhat. You'll have a MySQL Server, of course, but instead of the VFP IDE, you'll have your VFP executable and the VFP runtimes. Finally, you'll also have the ODBC driver, which your VFP application will rely on in order to connect to the database server.

MySQL Server

Installation and configuration of MySQL Server for production is a topic nigh unto itself. In some environments, you'll want to dedicate an entire box just for the database server, although MySQL is lightweight enough that many applications build it into their application and run it unbeknownst to the user. Nonetheless, the MySQL Server will be running on a separate box from the user's workstations; perhaps a file server, so that all of the company's data is sitting in the same place.

VFP Application (EXE, runtimes)

Installing and configuring your VFP application on end-user's Windows workstations for a client-server application is not much different than doing it for a traditional LAN application. In fact, there's no difference at all, except for the need to include the ODBC driver.

ODBC Driver

Each end-user installation of your application will require the inclusion of the ODBC driver as well. If you're using DSNs (instead of hand-crafting your own connection strings), those will need to be installed as well.

Deployment Scenarios – Locations

Server: MySQL Server

To sum up, the MySQL Server will be installed on a server machine, and that's all. No GUI tools, no VFP, no nothing. Ideally, once installed, you'll be able to rip the monitor, keyboard and mouse off the server and access the MySQL Server remotely from then on. Many of my MySQL applications run on a box I've never seen, under 5 miles of concrete and dirt deep in the Rocky Mountains.

Workstations: VFP Application (EXE, runtimes), ODBC Driver

You've been working with installing VFP apps on your end-users workstations for decades. With the exception of the ODBC driver, there's nothing new here. Given that installing an ODBC driver isn't simply a matter of copying a bunch of files into a directory, you'll probably need to start using an installation tool if you haven't already been using one.

Configuring MySQL

Concepts

MySQL is a service

Visual FoxPro and your applications are programs that run only as long as someone has started them. MySQL, on the other hand, is regularly configured as a service, so that it is running all of the time that the machine is on. As a result, MySQL is always sitting around, waiting for requests for data, much like a Web server waits for requests for Web pages.

Something connects to the MySQL service

Eventually, something connects to the MySQL service. That something might be a Visual FoxPro application, but it could be something else, such as a GUI tool like the MySQL Query Browser, or another application. The 'something' comes from a machine called a 'host'. In development, that host might be the same box – when you're running FoxPro and MySQL on the same machine. In a production environment, that host is going to be another machine. In both cases, the connection from the host includes username and password credentials.

MySQL authenticates via credentials database

Once the MySQL server receives a connection, it authenticates that connection's credentials – the host, the username, and the password – against a database of permitted username/password/host combinations. A set of sample credentials might look like this:

host	user	password
localhost	bob	secret
localhost	carla	verysecret


```
192.168.1.123    ted          moresecret
192.168.1.123    fredericka  topsecret
```

Bob and Carla can access the MySQL server only from the local box, while Ted and Fredericka can access the server only from a box on the local network with an IP address of 192.168.1.123.

Credentials have various permissions

These credential combinations also dictate the access to specific actions. For example, each set of credentials can be given (or denied) permission to select, insert, update, and delete records, create and drop tables, and so on.

Implementation

MySQL Monitor

The MySQL Monitor is the most useful tool for initially working with accounts and credentials. Fire it up in a DOS Command Box (Windows) or a Terminal Window (Linux/Mac), and you'll swear you're back at home with dBASE II:

```
mysql> _
```

mysql.user database

The initial installation of MySQL creates a database named 'mysql' that contains all of the system files. One table, users, contains accounts. Its structure looks like this:

```
mysql> select host, user, password, select_priv, insert_priv from mysql.user;
+-----+-----+-----+-----+-----+
| host      | user  | password      | select_priv | insert_priv |
+-----+-----+-----+-----+-----+
| localhost | root  | 38914f9560d4d960 | Y           | Y           |
| mybox     | root  | 38914f9560d4d960 | Y           | Y           |
| localhost |      |                  | Y           | Y           |
| mybox     |      |                  | Y           | Y           |
+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
mysql>
```

As you can see from this listing, MySQL creates four records by default. The first two are for the root user, and those accounts are secured with a password that you enter during installation. The other two are for the anonymous user, and these are not password-secured.

The passwords shown aren't the actual passwords, they're the hashes of the password.

Setting passwords and other account actions

Given that MySQL is a full-blown database server, you're going to need to do some database server administration before getting into your application development. First, you're going to want to set up security for users and hosts.

Appropriate users

The first thing you're going to want to do is get rid of those anonymous users. At the mysql> prompt, issue the SQL command

```
mysql> delete from user where user <> 'root';
```

If you didn't get around to providing a password for the root user, you can do so with the 'set password' function and the 'password' function:

```
mysql> set password for 'root'@'localhost' = password('secret');
```

You also need to add a work-a-day user that you'll use during development. Using the 'root' account is a bad idea for all sorts of reasons, the least of which not being the ridicule you'll face from others when they find out about your poor security practices.

```
mysql> grant all privileges on *.* to 'bob'@'localhost'
-> identified by 'secret' with grant option;
```

Tricks for hosts

As you can imagine, having your database accessible via a TCP/IP connection makes for loads of flexibility, but at the same time, you potentially expose it to unwelcome visitors. Using judicious values in the host column provides you with a significant level of security.

First, if you're running MySQL on your development box, there's no reason to have anything other than 'localhost' in the host column.

Second, if your MySQL server is on another box, it's ideal if you can explicitly delineate which remote boxes are allowed access. Typically, IP addresses are all you need:

host	user	password	select_priv	insert_priv
192.168.1.123	bob	38914f9560d4d960	Y	Y

Third, if your MySQL server is on your local network, but you have more than a machine or two, it can be a pain to enter the same user account for machine after machine. Instead, you can limit access just to other machines on the same subnet using a host wildcard like so:

host	user	password	select_priv	insert_priv
192.168.1.%	bob	38914f9560d4d960	Y	Y

Two Darn Handy MySQL Tools

There are a number of GUI Tools to help administer and access MySQL. Two of them come from the MySQL folks themselves, the MySQL Administrator (**Figure 6**) and the MySQL Query Browser (**Figure 7**).

MySQL Administrator

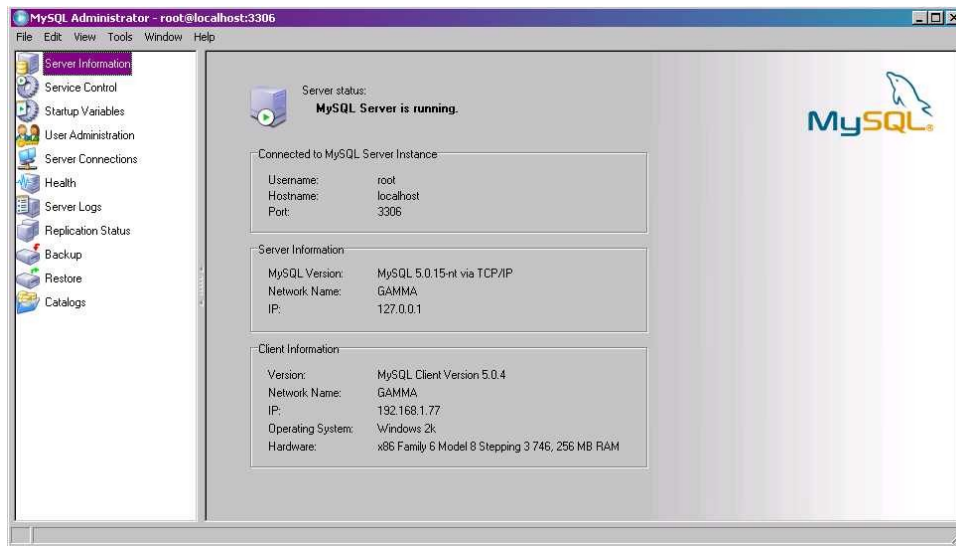


Figure 6. The MySQL Administrator allows point and click administration of the MySQL server.

The Administrator provides a single point of control for the entire server. Clicking on a node on the left side opens up a pane on the right with information and controls relevant to that function. Nodes include Startup Variables (configuration), User Administration, and Logs. You can also perform backup and restore, and configure replication via the Administrator.

You can even administer some functions remotely, while others (starting and stopping the server, for example) are only available on the local machine.

MySQL Query Browser

The Query Browser provides VFP Browse Window-like capabilities in an easy-to-use GUI. The Query toolbar and Edit text box at the top allow you to enter and execute commands and perform common commands. The Results area underneath is where results of SELECT commands as well as error messages are displayed. The Sidebar on the left side consists of the Object Browser (on the top) and the Information Browser (on the bottom.) The Object Browser's Schemata tab displays all databases and allows you to drill down into one, displaying all tables and columns in each table. The History tab displays recent commands executed in the Query Text box, providing a similar function to the command history in the VFP Command Window. The Information Browser provides quick access to MySQL command and function syntax.

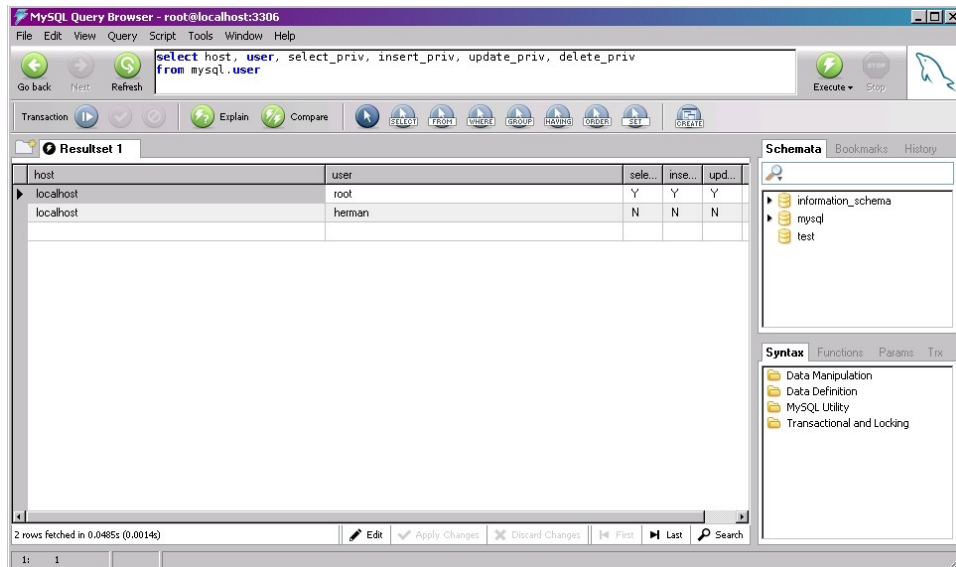


Figure 7. The Query Browser is a MySQL replacement for VFP's command window.

Not Darn Handy: Migration Toolkit

The Migration Toolkit available on the MySQL website is to be avoided. Limited in functionality, it requires Java and still appears to be a work-in-progress.

Connecting to MySQL from VFP Interactively

Now that the MySQL engine is set up and configured, it's time to get at it from VFP.

Installing the ODBC Driver

The ODBC driver, available on the MySQL Web site, is the doorway that provides access to MySQL from VFP. Installing it is a matter of a half-dozen mouse clicks. Once you do so, you'll see the driver in the ODBC Data Sources list, as shown in **Figure 8**.

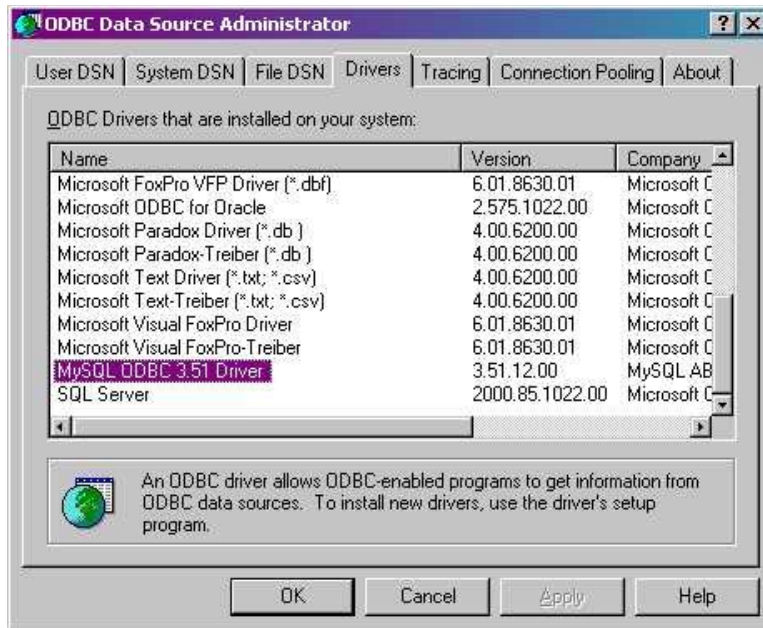


Figure 8. The Data Source Administrator shows all available ODBC drivers on the machine.

Connecting Via DSNs

DSNs are Windows objects that contain the information necessary to make a connection to a data source. These objects are stored in the Windows Registry. There are three types: File, User, and System.

A File DSN can be stored anywhere and used by any user who has access to the appropriate ODBC drivers. A User DSN, on the other hand, resides on the local machine and is tied to the User ID who created it. The third type, System, can be used by anyone who has access to the machine.

Open the ODBC Administrator from the Control Panel, select System DSN, select Add, select the MySQL driver, and you'll be presented with the Configure Data Source Name dialog as shown in **Figure 9**.

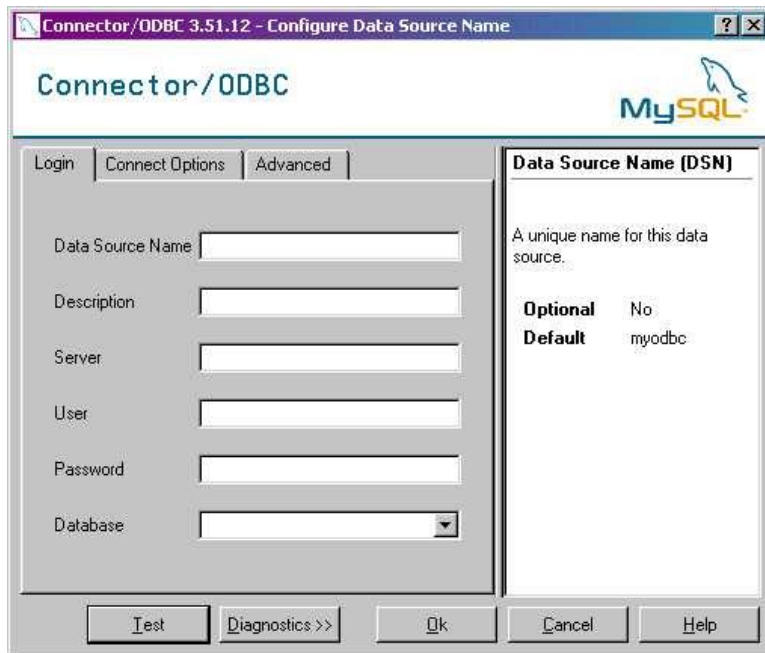


Figure 9. Configuring a DSN for MySQL.

Enter a name for the DSN ('sampledsnname', in the following examples), enter the server (either 'localhost' or the IP address of the MySQL server, such as 192.168.1.123), and the user name and password for the account.

Test the connection with the Test button (the Diagnostics button provides access to tools to troubleshoot failures) and then click OK.

Open up Visual FoxPro, and execute the following command in the Command Window:

```
m.liH = sqlconnect("sampledsnname")
```

The value of m.liH ('H' for 'handle') will be > 0 (starting with 1, and then incrementing by 1 for each successive connection) if the connection to MySQL succeeds.

Terminate just that connection by passing the handle to the sqldisconnect() function:

```
? sqldisconnect(m.liH)
```

If you've created multiple connections, each handle has its own value > 0. It can be a nuisance to terminate each one separately. You can terminate all connections by passing a '0':

```
? sqldisconnect(0)
```

Via Connection Strings

A DSN contains all of the information to connect to MySQL, and hides that from the user. You can also build a connection string manually and pass that string to MySQL from within VFP with the sqlstringconnect() function. Let's look at how a connection string is built.

Key-Value pairs

A connection string consists of multiple key-value pairs, like so:

```
UID=bob
```

The string “UID” on the left is the key and the string “bob” on the right is the value (or the variable, if you like.) A connection string consists of multiple key value pairs, each separated by a semi-colon.

Syntax

Here is an example of a complete connection string:

```
"DRIVER={MySQL ODBC 3.51 Driver};  
SERVER=localhost;  
UID=bob;  
PWD=secret;  
DATABASE=customers;  
OPTION=1"
```

You'll notice that there doesn't have to be a semi-colon at the end, any more than you need a comma at the end of a list of items. The entire connection string is enclosed with double quotes, in order to make it a string.

The syntax can be tricky, with spaces and zeros and 'el's all over the place, so here's a trick to assemble your first connection string. After creating a DSN, capture the values to the `_cliptext` variable, like so:

```
m.liH = sqlconnect("sampledsnname")  
_cliptext = sqlgetprop(m.liH, "ConnectionString")
```

Now you can paste `_cliptext` into a PRG or other handy location and examine the contents.

Using a Connection String Inside VFP:

Use the `sqlstringconnect()` function to connect to MySQL. (Most of the connection string has been removed to make the rest of the command clearer.)

```
m.liH = sqlstringconnect(  
    "DRIVER={MySQL ODBC 3.51....  
    ....secret;DATABASE=customer"  
)
```

Like `sqlconnect()`, `sqlstringconnect()` returns a value > 0 if successful. This value is the handle that you'll then use to send further communications over to the server.

Executing Commands Inside VFP

Once you have a handle, you can send SQL commands to the MySQL server. These commands include both DML (data manipulation language) commands like `SELECT` and `INSERT`, and DDL (data definition language) commands like `CREATE TABLE` and `ALTER TABLE`.

SQL SELECT:

The most simple command is a SQL SELECT on a single table that returns all of the rows. Note that you probably don't want to do this for anything more than a tiny (a few thousand rows) table.

```
m.liResult= sqlexec( ;
    m.liH, ;
    "select * from customer", ;
    "csrAllCust" ;
)
```

sqlexec() is the function that sends a SQL command to the MySQL server, using the handle already obtained. The third parameter, 'csrAllCust', is the name of the cursor that that results will land in. If you don't include a third parameter, MySQL will create a default cursor for you. The problem with doing so is that as soon as you issue a new SELECT, the cursor will be overwritten. If you specify the cursor name, then you have control.

SQL INSERT:

Similar to the previous example, you can send an INSERT command, like so:

```
m.liResult= sqlexec( ;
    m.liH, ;
    "insert into cust (cname, crep) ;
    values ('Bike World', 'Alice')" ;
)
```

Note that the third parameter is not included, as it doesn't have any place in an INSERT.

You can send any type of SQL command to MySQL this way – UPDATES, DELETES, CREATE TABLE, anything. The syntax, however, has to be legal MySQL SQL, not legal VFP SQL. That means if you're used to abbreviating VFP SQL commands to four letters, you'd better get out of the habit Real Soon Now.

Errors. Dreaded Errors.

In a perfect world, you'll never see an error. If you run across that perfect world, let me know. In the meantime, here's how to cope with the inevitable trouble. Suppose you haven't gotten out of the habit of abbreviating commands to four letters, and you include the following in your VFP program:

```
m.liR = sqlexec(m.liH, "sele * from x")
```

You're gonna see a big ol' -1 in m.liR. But why? After a few moments, this particular error may be evident, but others not so much. You can use aerror() to capture the error information of an errant SQL command, like so:

```
m.liR = sqlexec(m.liH, "sele * from x")
dime aOops[1]
? aerror(aOops)
```

Now, open up your Locals window in the Debugger, and you'll see the contents of aOops in all its glory. Note that aOops will be overwritten by the very next error that occurs, so you need to

deal with it right away. In my production systems, the next command after the `aerror()` line is a call to a generic error handler that saves all of the aOops values to a table, along with the values passed to the `sqlxec()` function.

Client-Server User Interfaces

Querying

As I mentioned at the beginning, your mind-set for accessing a client/server back end is different than the approach used with the traditional LAN application. In the olden xBase days, you designed your form as if you had access to the entire database (because, indeed, you did.) I always likened it to opening up a file cabinet with a huge drawer of file cards, and allowing the user to flip from the first to the second, the third, the fourth, and so on, back and forth, through the whole drawer.

Instead, you need to decide which records you want, bring those into your form, and provide navigation tools for that small subset. To carry the “file cabinet full of file cards” analogy forward, you’ll be grabbing just a handful of those cards (copies, actually) and putting them on your desk to work with.

A word of warning when you’re designing your UI; many developers use a small test data set. The reason is because they probably don’t have a realistic test database that contains millions of records and because, if they make a mistake (say, where the query DOES try to bring down the entire back end), they don’t want to be waiting until it finishes. Quite annoying.

The problem with this small data set is a mistake that causes you to hit the entire database won’t really be noticed. Keep in the back of your mind, “What if I had 37,000,000 records in here?” Use a small data set to perform functional testing, but then do some performance testing against a larger dataset. When your UI will handle those 37 million records with pretty much the same alacrity as the 105 that you have in your test database, then things are set up properly.

The logical and physical components

Back to the design itself. Your UI consists of two logical pieces – one that provides user-entered criteria (“filter parameters”) to select the desired records, and a second that allows the user to work with the result set. There are many ways to implement this in practice, and the differences really boil down to personal preference. Some people like to use a page frame, with the first page holding filter controls and the second (and, perhaps, subsequent) pages displaying the results. The second page might show a list of matches via a grid, and the third page would display the details of the record highlighted in the second page’s grid.

Another mechanism is to use two separate forms. The first form would accept filter parameters from the user. Entering a value (or values) then causes that form to close and open a second form that displays the results. This form might use a page frame with separate pages for results and details, similar to the first mechanism, or it might simply cram everything – results list and details – on the same form. This last design is the one I prefer, and will use in these examples.

As there are a lot of things going on here, I'm going to create this mechanism in a series of steps, each example building on the shoulders of the previous one. First, I'll create a single 'results' form that demonstrates the single page with both the results list and the detail fields crammed onto it. This form will use a local database as its datasource so we can concentrate on the design of the form, populating the results list, and then showing how navigating from one record to the next will update the controls for the detail fields. The second version will bring some complexity into play, showing how to display child records as you navigate from one record to the next in the results list.

The third version will add the filter criteria form to the mix. The filter form will only have one or two simple text fields; now isn't the time to build a complex query builder interface. The parameters entered into the filter form will be passed to the results form that will then winnow down the local database and create a smaller results set, mimicking the performance of the client-server mechanism we'll build in a moment. The big difference between this example and the previous one is that we're producing a small result set that the results form will work with.

Once you're comfortable with the structure of this two form interface, we'll swap out the code that handles the local data and replace it with code that connects to the MySQL back end and returns selected data. Since the operation of the rest of the form is based on the results set, this will simply be a matter of changing the method that populates the results set!

When you open the source code for the forms in this section (as well as subsequent sections), you'll see there isn't a lot of sophisticated subclassing, with objects flying all over the place. You're trying to learn how to build a UI for client/server, and see what the code underneath looks like, so that's what we're going to focus on. It can be terribly distracting if you're not too experienced with objects to the point that including that into the mix would be more confusing than helpful.

For example, a lot of folks like providing an independent toolbar for functions to control objects on a data entry form, such as navigation, add/edit/delete, search, and so on. I'm one of them. But the code that incorporates a separate toolbar with the form involves communication between the toolbar and the form – no, wait, make that the active form – determining which toolbar buttons need to be available... you see where I'm going with this.

Incorporating all of this toolbar code with your first query form just gets in the way of what we're trying to do, so I'll put all the necessary buttons right on the form. Same thing for separate 'biz objects' and n-tier programming models. That's too much to handle for the developer who is working with their first client-server system. We'll pass on these items for the time being as well.

Once you're generally comfortable with client-server philosophies and practices, consider picking up one of the frameworks that consists of a robust object-oriented set of data handling classes as part of the package if you want more sophistication. The purpose of this article is to provide a simple enough set of code to teach the principles of client-server development without a robust infrastructure. The purpose of such a framework is to provide that infrastructure.

Source code

The source code for this article is contained in two files, `common.zip` and `query.zip`. You can use these files in one of two ways. The first is to create a directory, like `'mysql\query'`, and unzip

both files into this directory. Then open VFP and change the default directory to the directory you unzipped the files into, like so:

```
cd \mysql\query
```

When you open the forms, you'll probably be prompted to look for the hwctrl.vcx (and other) class libraries. Just point to the directory you put them in.

The second way is to create separate directories for common files and the source code for this chapter, like so:

```
e:\mysql\common  
e:\mysql\query
```

Then run VFP, change the default directory to \mysql\query, and set your path to include the common directory:

```
cd \mysql\query  
m.lcOrigPath = set('path')  
m.lcNewPath = m.lcOrigPath + iif(empty(m.lcOrigPath), "", ",") ;  
    + "\mysql\common"  
set path to (m.lcNewPath)
```

The common.zip file contains program files that are used in more than just these examples e.g., procedure files, base classes, and graphics. The query file contains all of the program files and database files needed for the examples in this section.

Base classes

(hwctrl.vcx)

All of the forms in these examples are built using the same set of base classes found in hwctrl.vcx. These classes included a sub-classed version of the visual VFP controls we'll need for these examples, and a set of form classes. These subclasses range from very simple, with only a couple of custom properties and methods each, to fairly simple, with the bare minimum of properties and methods needed to get the job done.

Here is the nickel tour of the classes we're going to use for this chapter's examples.

The forms that display results are all based on the hwfrmnav class. This class is based on the hwfrm form class, sub-classed from the VFP form base class. While the hwfrm class has some new methods and properties, none of them are germane to this specific example, so I'll bypass a detailed explanation. However, the hwfrmnav class has some very important modifications.

First, it has several controls that are included by default. The Done button automatically calls the form's Close() method. The two textboxes with yellow backgrounds are what I call 'developer-only' controls. They're only displayed on the form when the form is run interactively (in the VFP IDE), or when a special 'run-as-developer' parameter is passed to the application

during test or production use. I typically use these ‘developer-only’ controls to display information such as primary key values or other pieces of data used during testing or debugging.

The listbox on the hwfrmnav class comes from the hwlstnav class, and has two very important items – a custom property and a custom method. The `aItems[1]` property is used to populate the list; `RowSourceType` is set to 5-Array, and `RowSource` is set to `this.aItems`. Thus, in order to fill the listbox, `aItems` just needs to be populated. The custom `AnyChange` method is called from both the `InteractiveChange` and `ProgrammaticChange` methods of the native class. All code that is run when the list is used is put in `AnyChange`, instead of having to duplicate it in both the `IAC` and `PC` methods.

Simple navigation form (*simplenav.scx*)

The first version of the form, *simplenav.scx*, contains just a listbox used for navigation and standard controls to display data. Its purpose is to display the businesses and locations in the `INS` database, and allow the user to navigate from one to another, much like they would in the aforementioned file cabinet by scrolling up and down through the listbox. See **Figure 10**.

Figure 10. A simple form using a listbox as a navigation tool.

This form is based on a supporting cursor named `csrRes` that contains the result set that will populate the controls in the form. This cursor can be created from a single table, or from the join of multiple tables. The important point is that the result set consists of a limited number of records.

Identifying fields from the supporting data set are loaded into the listbox, allowing the user to scroll through the table and display all fields for the record in the controls on the right side of the form. Under the hood, the `init()` method calls `GetResults()` that creates the `csrRes` cursor. In this first incarnation, `GetResults()` simply runs a SQL `SELECT` to create a join of businesses and locations, and then populates the listbox’s `aItems` array with a user-friendly list of business names.

The `AnyChange` method of the listbox contains the following code

```
select csrRes  
go this.ListIndex  
  
thisform.TableToForm(this.ListIndex)
```

Since there is a one-to-one relation between the listbox's array and the supporting cursor, any action in the listbox, be it interactive or programmatic, will cause the record pointer in the csrRes cursor to be moved to the record that matches the selected array element.

If you're thinking that you wouldn't want to use this interface to navigate through a cursor that had 37,000,000 records, you're right, both technically and practically. Technically, versions of VFP before 9.0 can't support an array that big – it's limited to 65,000 elements in an array, and even with 'just' that many, it'd likely be slow going. (VFP 9 supports two GB.) In a practical sense, how would you scroll through a listbox that contains thousands and thousands of rows? This is why we're going to use a filter to winnow down the number of records in the result set.

Once the record pointer is moved, the form's custom TableToForm method is called with the record number of the newly selected record. TableToForm is a centralized location for all the code that is involved in displaying the record's data in the form's controls. (If you're wondering why this wasn't done by simply setting the ControlSource of each control to the underlying cursor, the answer is that in this simple example, it could have been. However, we're shortly going to run into situations where the mapping won't be as clean as one field to one control. Thus, we introduce this mechanism now, and will expand on it in subsequent examples.)

Simple navigation with children form

(*simplenav_withchild.scx*)

The second version of the form, simplenav_withchild.scx, expands on the first example by displaying child records for the business-location results set in a pair of listboxes. The first listbox shows the contacts for a specific location – voice and fax phone numbers, website URL, and so on. The second listbox shows the categories that the business belongs to. It is a listbox, as some organizations can be grouped under more than one listing. See **Figure 11**.

Figure 11. A simple form with child records displayed in listboxes.

This form has several changes. The obvious ones are a larger form area and the addition of two listboxes for the child records. Under the hood, the TableToForm method has been enhanced to populate both listboxes. (Told you we were going to need it!) Upon moving to a new record in the result set, the TableToForm method grabs the associated child records based on the primary key in the csrRes cursor, and populates the listboxes with the results.

Simple navigation with filter

(*simplenav_withchild_andfilter1/2.scx*)

Now it's time to skinny down that list of records in the result set. We do this with a separate filter form that collects the parameter and passes it on to the results form. See **Figure 12**.

Figure 12. The Filter form allows the user to enter a value to select a subset of records.

Once a filter value has been entered and the user selects Find, the filter form is closed and the results form is opened. However, contrary to what you might think, the filter value isn't passed as a parameter. Instead, an object reference to the calling form is passed. In the GetResults() method, this object reference is used to determine the value of the filter value. While it's true that it would be easier to simply pass the name of the business in this example, what happens when you have three, four, five, or more filter values that have to be passed to the results form? Passing a whole string of parameters, some of which may not exist, quickly becomes a mess; it's much

easier and cleaner to pass a single object reference whose utility will grow as the filter form gets more complex. And applications always get more complex!

Note: Andy Kramek interjects here: Once you're comfortable with this technique of passing an object reference instead of a string of parameters, consider the use of a parameter object, where each property's name is the name of a parameter, and the values in the properties are the values to be passed. This technique avoids the tight-coupling of these two forms (what happens when the name of one of the controls on the filter form changes? Chaos!), uses fewer system resources, and provides better encapsulation.

Now onto the results form. The layout of the form doesn't have to change at all; indeed, nothing changes except the contents of two methods.

The `init()` method of the results form now includes a line to accept the filter form's object reference parameter:

```
thisform.oCallingForm = toCaller
```

The `GetResults()` method has two changes. The first is the determination of the filter value on the filter form:

```
m.lcBusinessName = alltrim(thisform.oCallingForm.hwtxtBusinessName.value)
```

and the second is a modification to the SQL `SELECT` statement, so the filter value is actually used in the query:

```
select biz.iidbiz, biz.cnabiz, ;  
    loc.iidloc, loc.cstreet, loc.ccity, loc.cstate, loc.czip, lIsHQ, tClosed ;  
from BIZ, LOC ;  
where BIZ.iidbiz = LOC.iidbiz ;  
and BIZ.cNaBiz like m.lcBusinessName + "%" ;  
order by biz.cNaBiz, loc.cStreet ;  
into curs csrRes
```

I know I just said that the UI of the form doesn't have to change, but if you looked at the new example in the source code, you'll see a new control – a read-only text box positioned above the list box. See **Figure 13** if you haven't opened the source code yet.

Figure 13. The results form now includes a read-only textbox that displays the Filter criteria.

This control will contain the filter expression used to populate the result set. I think it's a courtesy to the user to show them what expression was used to produce the results they're looking at. In the `GetResults()` method, I grabbed the value of the label of the filter value text box and appended the value entered in the text box to create the filter expression. I used the value of the label instead of the field name, figuring that "Business Name" was easier for the user to understand than, say, "cNaBiz".

Run the form and enter a single letter of the alphabet, such as "m" or "a." The listbox in the result form displays just those matching records. Or try entering a full name, like "Baskin-Robbins" or "Pizza Hut."

The last change made to this form is in the `close()` method. The filter form is closed, meaning it's no longer visible. However, the results form still has a reference to it. To show this, open the Locals debug window, and then run the filter form. You'll see an object variable named "simplenav_withchild_andfilter1:" if you click on the plus sign to the left, the properties of the object displays. When you eventually close the results form, you'll still see the variable "simplenav_withchild_andfilter1" displayed in the Locals window. This is a 'dangling reference' to an object that's been destroyed, and this can be bad; in the worlds of IBM in the days of Lotus 1-2-3 and the original IBM PC, "unexpected results may occur." Here, those results usually consist of C5 memory errors at the most inopportune times.

To get rid of the reference, add the following code

```
release simplenav_withchild_andfilter1
dodefault()
```

to the `Close()` method of the results form.

Simple navigation with MySQL backend

(*simplenav_withmysql1/2.scx*)

Now it's time for the main event – hooking up our filter and results forms to talk to MySQL instead of local DBFs. Hold on tight! In this example (and the next), we use the INS MySQL database included in the source code, found in INS.ZIP. (This is a simple MyISAM set of tables that can be placed in your MySQL data directory.)

There are four separate changes that need to be made to the *simplenav_withchild_andfilter* forms. (1) a connection to the database needs to be made (and disconnected when the form is closed), (2) the resulting handle needs to be stored as part of the form, (3) the *GetResults()* method needs to get its results from the back-end instead of from local DBFs, and finally, (4) the *TableToForm()* method needs to be updated to handle child SELECTS.

Because we're connecting to the INS database via a connection string, we need to add one more change that allows the host, username, and password to be passed to the filter form via parameters, just like in examples in earlier chapters. Thus, we call the filter form like this:

```
do form simplenav_withmysql1 with 'localhost', 'bob', 'secret'
```

but change the parameter values as appropriate for your system. The filter form looks the same as the one shown in Figure 13, but the results form has been slightly modified as shown in Figure 14.

The screenshot shows a software window titled "Business Locations, Contact Information & Biz Types (Handle: 1)". Inside the window, there is a "Filter:" field containing "Business = m%". Below the filter is a list of business names: "M Quest Clothing", "M&I Bank", "Mark B Smuckler MD", "Maureen Stalle' Team, Realty Exec Lakeshore", "Modern Touch Dental" (which is highlighted), "Moon, Laurence M", "Moore, Jordan A MD", and "Murray's Wine Spirits". To the right of the list, there are several input fields: "Business" (Modern Touch Dental), "Street" (105 W Silver Spring Dr), "City" (Whitefish Bay), "State" (WI), "Zip" (53217), "Contacts" (voice 414.964.0680), and "Type of Biz" (Dentist). At the bottom, there is an "Out of Business" field with "...", two yellow buttons labeled "idbiz" and "idloc" both with "105" next to them, and a "Done" button.

Figure 14. The results form now includes the handle of the database connection in the title.

If you're scratching your head, trying to figure out what the difference between Figures 13 and 14 are, look in the title bar of the form.

Pass connection parameters to simplenav_withmysql1

First, we need to add three properties to this filter form. These properties hold the values that are passed into the form so they can be accessed from the results form. The results form is where we're making actual connection to the database, but we enter the 'system' from the filter form, so that's where we need to send in the connection parameters.

The properties are named cdbhost, cdbun, and cdbpw. All character strings, all database related, all initialized to be empty strings.

Next, we need to modify the init() method of the form to accept parameters, and to store them to the form's properties:

```
lparameters m.tcdbhost, m.tcdbUN, m.tcdbPW
if pcount() < 3
    messagebox("You must pass the host, username & password as a parameter:" ;
        + chr(10) + chr(13) + chr(10) + chr(13) ;
        + "do FORM with 'localhost', 'bob', 'secret'")
    return .f.
endif
thisform.cdbhost = m.tcdbhost
thisform.cdbUN = m.tcdbUN
thisform.cdbPW = m.tcdbPW
dodefault()
```

Notice that the init() returns false if less than three parameters are sent in; this prevents the form from instantiating. No sense in teasing the user, is there?

Finally, the Find button's click() method has been tweaked to call the new results form:

```
do form simplenav_withmysql2 with thisform
```

Now let's look at the results form, simplenav_withmysql2.scx.

Store connection handle in simplenav_withmysql2

Once we create a connection, it needs to be stored as a property of the form so it can be referred to during various database operations. The property, iHandle, is this property. It's initialized to be a numeric value.

simplenav_withmysql2.init()

The init() method of the results form needs to set up the connection before any work with the database itself is performed. Init() looks like this:

```
thisform.cdbhost = alltrim(thisform.oCallingForm.cdbhost)
thisform.cdbUN = alltrim(thisform.oCallingForm.cdbUN)
thisform.cdbPW = alltrim(thisform.oCallingForm.cdbPW)
```

```

if thisform.CreateConnection()
    if !thisform.GetResults()
        return .f.
    endif
else
    return .f.
endif

```

Since the database work starts with the GetResults() method, init() needs to call the CreateConnection() method just before the GetResults() method. If CreateConnection() succeeds, GetResults() is called next, and if that also succeeds, init()'s job is finished.

Now, what if the attempt to create the connection fails? We need to provide feedback and not open the results form – what would be the point? If the connection fails, we have the CreateConnection() method return a .f. back to its caller, the form's init() method.

The init() method didn't simply just call CreateConnection() blindly. Instead, it tests the return value from the method, and if .f. is returned, init() in turn returns .f., which prevents the form from instantiating. The CreateConnection() method has already displayed an error dialog to the user, so init() doesn't have to perform that chore.

GetResults() works through the same mechanism. If GetResults() fails (we'll see the code in a moment), GetResults() also returns a .f. value, which causes init() to return a .f., and, again, the results form does not instantiate. As an aside, the definition of failure of GetResults() would be due to bad SQL statements, for example, or a back-end database problem – an empty result set is not a failure.

CreateConnection()

The CreateConnection() method looks like this:

```

m.liH=sqlstringconnect( ;
+ "DRIVER={MySQL ODBC 3.51 Driver};" ;
+ "SERVER="+thisform.cdbhost+";UID="+thisform.cdbUN ;
+ ";PWD="+thisform.cdbPW+";database=INS")
if m.liH < 1
    messagebox("No connection; handle is:"+transform(m.liH)+":")
    return .f.
else
    thisform.iHandle = m.liH
endif
thisform.Caption = alltrim(thisform.Caption) ;
+ " (Handle: " + transform(thisform.iHandle) + ")"

```

The last line in this code snippet shows the modification of the caption in the title bar to reflect the handle.

GetResults()

Let's look at the changes made to GetResults() now. First, the way you build search expressions to send to MySQL is different. You need to add a "%" to the end of the filter expression, like so:

```
m.lcBusinessName = alltrim(thisform.oCallingForm.hwtxtBusinessName.value) +  
"%"
```

and then use 'like' in the SQL SELECT command. We're going to use the "text to m.lcStr" command to build the m.lcStr string that's going to be sent to MySQL via SQLEXEC(). The SQL SELECT command itself also changes, because the fields in the MySQL LOC table are different from the DBF version.

```
text to m.lcStr textmerge noshow pretext 15  
select biz.iidbiz, biz.cnabiz,  
    loc.iidloc, loc.cno, loc.edir, loc.cstreet, loc.csuf, loc.csecline,  
    loc.ccity, loc.cstate, loc.czip,  
    iishq, tclosed  
from BIZ, LOC  
where BIZ.iidbiz = LOC.iidbiz  
and BIZ.cNaBiz like '<<lcBusinessName>>'  
order by biz.cNaBiz, loc.cStreet  
endtext
```

Next, there needs to be two levels of trapping in the GetResults() method, as opposed to just one when we were dealing with local DBFs. The first level of trapping concerns whether the SQLEXEC() function actually worked or not. The following code snippet shows that if SQLEXEC() doesn't work, the error is recorded via aerror(), the user is notified, and then GetResults() returns .f. This sends the function back to the init(), which in turn returns a .f., thus preventing the results form from opening. The pseudo-code looks like this:

```
m.liX = sqlexec(thisform.iHandle, m.lcStr, "csrRes")  
if m.liX > 0  
    * code for success  
else  
    * record error with aerror  
    * notify user with messagebox()  
    * then return .f.  
endif
```

The details of the ELSE branch of the IF construct look like this:

```
local aOps[1]  
m.liHowManyErrors = aerror(aOps)  
for m.li = 1 to m.liHowManyErrors
```

```

insert into ZOOPS ;
(inoerr, ctext, ctextodbc, csqllstate, inoerrsql, ihandle, tadded) ;
values ;
(aOps[m.li,1], aOps[m.li,2], aOps[m.li,3], aOps[m.li,4], ;
aOps[m.li,5], aOps[m.li,6], datetime())
next
* display nothing
messagebox("SQL Execution failed; code is" ;
+ chr(10) + chr(13) + chr(10) + chr(13) ;
+ transform(m.lcStr))
return .f.

```

The use of the `aerror()` function and the recording of the errors isn't anything new – it's nearly verbatim the same error trapping code shown in Chapter 12. However, after the error(s) have been logged, note the messagebox that notifies the user about what happened, gives them the string of code that failed, and then returns `.f.` to the calling function. An alternative to displaying the code to the user is recording it in the ZOOPS error logging table.

The second level, built into the 'success' side of the '`m.liX > 0`' test, determines whether or not any result sets have been returned. If `> 0`, at least one result set has been returned, so check to see if there are any records in it, and proceed from there.

```

if m.liX > 0
* display
if used('csrRes') and reccount('csrRes') > 0
m.liNumRows = reccount("csrRes")
* fill array populating the listbox
else
* no records in csrRes
* display nothing in array
endif

```

The empty `csrRes` cursor is a valid occurrence, so we don't want to return `.f.` from the ELSE branch of the IF construct. For example, if the user entered "ZYX-o-plenty," it would be reasonable to assume there might not be any matches in the INS database. Well, at least around here, there aren't any businesses with that name.

That takes care of the `GetResults()` method. Now we turn our attention to the `TableToForm()` method.

TableToForm()

The `TableToForm()` method is where we populate the detail controls on the Results form. There are two places where changes have to be made.

The first is where individual controls are assigned values from fields in csrRes. The fields in the MySQL LOC table don't map directly to the LOC DBF fields – the MySQL structure has individual attributes of a street address broken up into separate fields. Thus, those attributes have to be reassembled to be displayed in the 'Street' textbox. Trivial, but needs to be done:

```
thisform.hwtxtStreet.Value ;
  = alltrim(csrRes.cno) + " " ;
  + alltrim(csrRes.edir) + " " ;
  + alltrim(csrRes.cStreet) + " " ;
  + alltrim(csrRes.cSuf) + " " ;
  + iif(empty(csrRes.cSecLine), "", ", ") ;
  + alltrim(csrRes.cSecLine)
```

The second change is more involved, not conceptually, but in a busy-work kind of way. The child listboxes, for Contacts (populated from the COOR table) and for Business Types (populated from the ZLOOKUP table via the BIZCAT join table) need to be filled with data from the MySQL tables. This means more SQLEXEC() functions using modified versions of the SQL SELECTS that were used with local DBFs.

The code to dig out the contacts for the business highlighted in the Business listbox on the left of the Results form looks like this:

```
text to m.lcStr textmerge noshow
select iidcoor, cdata, etype from COOR
  where coor.iidloc = <<thisform.hwtxtdevIID2.value>>
endtext
m.liX = sqlexec(thisform.iHandle, m.lcStr, "csrResCoor")
```

If the SQLEXEC() function succeeds, the Contacts list box is populated by using the following code:

```
if m.liX > 0
  m.liNumRows = reccount("csrResCoor")
  if m.liNumRows > 0
    dimension thisform.hwlstCoor.aItems[m.liNumRows,2]
    m.li = 1
    scan
      thisform.hwlstCoor.aItems[m.li,1] ;
        = csrResCoor.eType + " " + csrResCoor.cData
      thisform.hwlstCoor.aItems[m.li,2] = csrResCoor.iidCoor
    endscan
  else
    * code to populate the list box if there were no COOR records
  endif
```

There's one subtle difference in the line of code that fills the first column of the `hwlstCoor.aItems` array. In the local DBF version – Simple Navigation with Children – the first column is created by simply concatenating the contact type (such as 'voice') and the contact data (such as the phone number) like so:

```
thisform.hwlstcoor.aItems[m.lii,1] = aCoor[m.lii,3] + aCoor[m.lii,2]
```

This produces a list box that looks like this:

```
voice      414.555.1212
fax        414.555.1213
web        www.example.com
```

because the contact type data is stored in a fixed length field, which means there will always be padding out to the full length of the `cType` field. However, note that the type of contact in the MySQL COOR table is an enumerated field, which means there isn't any padding in the data in MySQL. Thus, the VFP cursor field representing the enumerated field data will be as wide as the longest field value, with no padding. The listbox would look like this:

```
voice414.555.1212
fax  414.555.1213
web  www.example.com
```

So we'll need to add a space, like so:

```
thisform.hwlstCoor.aItems[m.li,1] = csrResCoor.eType + " " + csrResCoor.cData
```

Note also that the field name has changed from `cType` in the DBF to `eType` in MySQL.

The rest of the code in the `TableToForm()` method is straightforward, featuring the same error trapping for failure of `SQLEXEC()` as well as testing for whether the resulting cursor has no records.

Close()

The last change in the `simplenav_withmysql2.scx` form has to do with closing the form. We want to make sure to close the connection. This is done like so:

```
m.liResult = sqldisconnect(thisform.iHandle)
debugout iif(m.liResult = 1, ;
    "Successful disconnect", "Unsuccessful disconnect")
release simplenav_withmysql1
dodefault()
```

You may notice that I have a debugout statement indicating success or failure – it doesn't take any appreciable amount of time, but can be handy while debugging during interactive development.

The purpose of this chapter is to create a mechanism for simple writing operations – add/edit/delete. We’re going to build separate forms to demonstrate adding, editing, and deleting. Then we’ll build a fourth form that puts all three functions together, and throws in the handling of multiple tables as well.

First, though, we need to do some housekeeping with the functions that will support these forms.

Creating a class library for client-server functions

Until now, we used a procedure file, `z_sql.prg`, for common functions. Let’s move those into a simple class library.

The `hwlib.vcx` class library already contains the `hwlib` class. This class has a number of general purpose functions, such as a function that reads INI files. We’re going to add a second class called “`cslib`” that holds functions specific to client-server applications.

First, create another class in our `hwlib` class library like so:

```
create class cslib of \common\hwlib.vcx as custom
```

Then add three methods:

- `z_tfed` (test for empty date)
- `z_es` (escape string)
- `z_sqlerror` (insert SQL error info into ZOOPS)

The class library is instantiated, say, as an object called `ocslib`, like so:

```
ocslib = newobj("cslib", "hwlib.vcx")
```

Then, calls to these functions will look like this

```
ocslib.z_es("some string")
```

or

```
ocslib.z_tfed(a date)
```

Since this class will only be used with client-server forms, we’ll instantiate the class inside the form. Where? If we were going to bind controls to data, we could do this in the `Load()` method so the class is available as controls are instantiated. On the other hand, if we’re going to need to pass parameters to the class, this would need to be done in the `Init()`. We’ll put it in the `Load()`. Details to come.

If you’re following along by running the samples provided in this chapter’s source code files, I suggest you open your Locals window so you can watch variables, including object references, appear and vanish when the forms are created and destroyed.

Building a simple form to add records

(*ui_add.scx*)

As this is our first form, it will be very simple – a couple of text boxes and a couple of Save buttons. The purpose is to add records to the BIZ table – a perfect choice since it only has two fields of interest – cNaBiz and cSecLine. The completed form is shown in **Figure 15**.

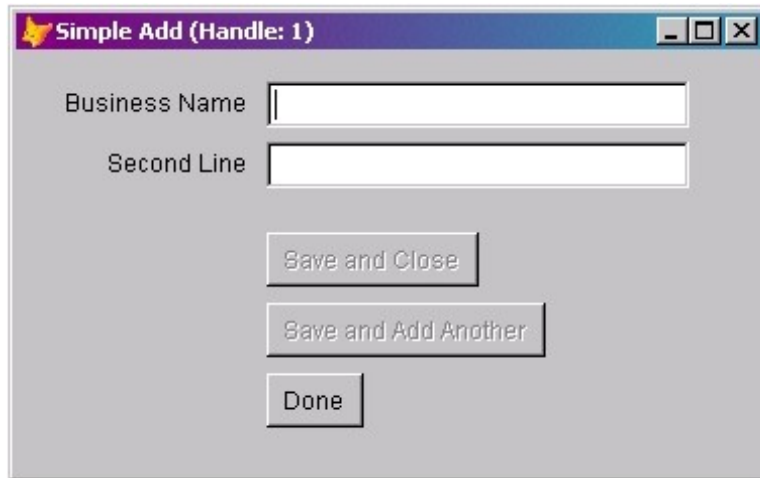
The image shows a Windows-style dialog box titled "Simple Add (Handle: 1)". It has a purple title bar with a yellow star icon on the left and standard minimize, maximize, and close buttons on the right. The main area is light gray. There are two text input fields: the first is labeled "Business Name" and the second is labeled "Second Line". Below these fields are three buttons stacked vertically: "Save and Close", "Save and Add Another", and "Done".

Figure 15. A simple form for adding new records.

User interface

The user calls this form by passing the hostname, username, and password, like so:

```
do form ui_add with 'localhost', 'whil', 'secret'
```

and the cursor is positioned in the “Business Name” text box.

Once the user enters a Business Name and leaves the field (probably by tabbing out), the “Save and Close” and “Save and Add Another” buttons are enabled. The first button commits the values in the two controls to a new record in the table and closes the form, while the second adds the record, blanks the text boxes, and then repositions the cursor to the Business Name text box so the user can quickly add another record.

The “Done” button acts as a “Just Kidding” release for those times when the user opens the form but then doesn’t actually want to add a record. Because this is an example, this doesn’t do any fancy validation such as searching for duplicate business names or attempting to correct spelling or capitalization of the data.

Internal design

Under the hood, this form is based on the hwfrm class. If you’re building this yourself, create the form like so:

```
create form ui_add as hwfrm from hwctrl.vcx
```

Then add the properties `ih` and `ocslib` to the form, and include the code for the `Init()` method listed shortly. The `Init()` method is overridden with the code that accepts the hostname, username, and password as parameters and attempts to connect to the database with them. If it's not successful, the user is notified and the form is not instantiated; if successful, the connection handle is assigned to a property of the form.

Next the `cslib` class is instantiated and a reference to the `ocslib` object is assigned to a form property as well. Finally, the caption is modified to reflect the connection handle – this last is purely for our purposes during development.

```
* ui_add.init()
lparameters m.tcdbhost, m.tcdbUN, m.tcdbPW
debugout this.Name + '.Init'
if pcount() < 3
    messagebox("You must pass the host, username & password as parms, like so:" ;
        + chr(10) + chr(13) + chr(10) + chr(13) ;
        + "do FORM with 'localhost', 'bob', 'secret'")
    return .f.
endif
m.liH=sqlstringconnect( ;
    + "DRIVER={MySQL ODBC 3.51 Driver};" ;
    + "SERVER="+m.tcdbhost+";UID="+m.tcdbUN+";PWD="+m.tcdbPW+";database=INS")
if m.liH < 1
    messagebox("No connection; handle is:"+transform(m.liH)+":")
    return .f.
else
    thisform.iH = m.liH
endif

dodefault()

ocslib = newobject("cslib", "hwlib.vcx")
thisform.ocslib = ocslib

thisform.Caption = alltrim(thisform.Caption) + " (Handle: " +
transform(thisform.iH) + ")"
thisform.hwlblSaveFailed.Caption = ''
```

You can run the form, `ui_add`, as is even before adding any controls. You'll get a blank form because there are no controls on it. However, if you have the Locals window open, you'll see the `olib` and `ui_add` objects created. `olib` comes from the `Init()` of the `hwfrm` class. If you drill into the `ui_add` object, you'll see the `ocslib` object reference. When you close the form, you'll see both objects destroyed, because the form is no longer in scope.

In order to clean things up, add the following code to the `close()` method:

```

* ui_add.close()
m.liResult = sqldisconnect(thisform.iH)

debugout iif(m.liResult = 1, "Successful disconnect", "Unsuccessful
disconnect")

dodefault()

release ui_add

```

This code makes sure the connection is closed and releases the reference created to the form itself. Try closing the form and watching the Locals window, before and after adding this code to the close() method.

Now let's add controls to the ui_add form. Text boxes named hwtxtBusinessName and hwtxtSecondLine, command buttons named hwcmdSaveAndClose, hwcmdSaveAndAddAnother, and hwcmdDone, and a label named hwlblSaveFailed. The LostFocus method for both text boxes calls the custom validate_data() method:

```

* ui_add.validate_data()
if empty(thisform.hwtxtBusinessName.value)
    thisform.hwcmdSaveAndClose.Enabled = .f.
    thisform.hwcmdSaveAndAddAnother.Enabled = .f.
else
    thisform.hwcmdSaveAndClose.Enabled = .t.
    thisform.hwcmdSaveAndAddAnother.Enabled = .t.
endif

```

validate_data() is what ensures that there is data in the Business Name text box before the Save command buttons are enabled.

Saving data

The Click() method in both Save buttons runs code that calls the form's Save() method, and then traps for success or failure in the method's return value. The two methods are different because one shuts down the form while the other clears out the text boxes, disables the Save buttons after a successful save, and repositions the cursor in preparation for another entry.

```

* ui_add.hwcmdSaveAndAddAnother.click()
with thisform
if .save()
    .hwtxtBusinessName.Value = ""
    .hwtxtSecondLine.Value = ""
    .hwtxtBusinessName.SetFocus()
else
    .hwlblSaveFailed.caption = "An error has happened during save. Please see the
error log."
    .hwcmdsaveAndAddAnother.Enabled = .f.

```

```

        .hwcmdsaveandClose.Enabled = .f.
endif
endwith

```

The form's Save() method, while we're talking about it, is fairly straightforward for this simple form, but it's a good place to get our feet wet.

```

* ui_add.save()
* save data
* iidbiz is auto-incr field in INS.biz
m.lcNaBiz = thisform.ocslib.z_es(alltrim(thisform.hwtxtBusinessName.Value))
m.lcNaSec = thisform.ocslib.z_es(alltrim(thisform.hwtxtSecondLine.value))
text to m.lcStr textmerge noshow pretext 7
    insert into INS.biz
        (cnabiz, cnasec, cadded, tadded, cchanged, tchanged)
    values
        ('<<m.lcNaBiz>>', '<<m.lcNaSec>>',
        'bob', now(), 'bob', now() )
endtext
m.liX = sqlexec(thisform.iH, m.lcStr)
if m.liX > 0
    * continue on
    return .t.
else
    local aOps[1]
    m.liHowManyErrors=AERROR(aOps)
    =thisform.ocslib.z_sqlerror(@aOps, m.lcStr, m.liHowManyErrors)
    return .f.
endif

```

After sending the contents of both text boxes through our z_es() escape string function, the resulting values are used in the new record being added via a SQL INSERT command. A point worth mentioning, because it'd be easy to miss, is that the primary key for the BIZ table is auto-incremented on the server's end, so our INSERT command doesn't need to specify the iidbiz field at all.

From our work in previous chapters, you should recognize the "text to ... textmerge" command as well as the <<>> delimiters. Errors are saved up via the z_sqlerror() call if the SQLEXEC() function call fails.

If you want to play around with this a bit more, try adding validation of the Business Name, ensuring that it is unique, before committing it to the table.

A simple two table edit form

(*ui_edit.scx*)

Each example we do gets a bit more complex. This edit example brings a second table into the mix. It provides the ability to navigate through a list of businesses and their locations (using the form created in the last section), and allows edits (and saves!) to both the business info and the locations. For this example, we're going to ignore the BizTypes, Coordinates, and People child list boxes. **Figure 16** shows what our finished form looks like.

Figure 16. A simple form for editing existing records.

User interface

Even a simple example like this provides the opportunity for a lot of design decisions, and in order to focus on the meat of the matter – editing and saving data back to a MySQL back end – I took the easy way out in terms of UI so as to not distract us. I didn't include all of the tiny bits of polish that a production version of this form would have. I'll mention a couple of these shortcuts as we develop the form, and we'll add those in down the road with future examples.

The first shortcut is that we're going to load up the list box with all of the records in the business and location tables. There are only a few hundred records in this sample database, so doing a SELECT ALL won't pose much of a burden on the form. This enables us to avoid the use of an initial form to filter the database first. This means we call this form just like the `ui_add` form:

```
do form ui_edit with 'localhost', 'whil', 'secret'
```

passing parameters that are handled in the `init()` of the form.

The corresponding data for the currently highlighted row in the list box displays in the controls to the right, just like in our navigation form in Chapter 18. In this form, however, the

user can edit a value in one of the controls; once they tab out of the control, the Save button is enabled if they changed the data.

As the user highlights businesses in the list box, the values in the controls on the right side are displayed for the selected business in the list box. The list box displays values from the denormalized cursor, `csrRes`, which contains data from a join of the BIZ and LOC tables. As you see in Figure 16, it is possible to see the same business name more than once, each instance representing a different location.

This denormalized architecture makes it easy for the user to find a business (and we'll expand on this in future examples to allow searching by street name), but it makes the construction of the underlying code a bit more difficult. The user can edit either the Business Name/Second Line values from the BIZ table, or any of the attributes in the LOC table, and not be aware that there is more than one table under the hood supporting the form. However, when the user saves a record, we have to figure out which (if not both) tables need to be updated. But we're database jocks, so I suspect we can handle this little chore.

The form also has a Done button that simply closes the form (after attending to other chores, such as closing the connection and cleaning up object references). For the time being, uncommitted changes are just abandoned.

Internal design

Under the hood, this form is based on the `hwfrmnav` class. If you're building this yourself, create the form like so:

```
create form ui_edit as hwfrmnav from hwctrl.vcx
```

Again, add the properties `ih` and `ocslib`, and include the same code for the `Init()` method as for the `ui_add` form. Then make one simple change to the `Init()` code. In the `ELSE` clause of testing for connection success, call the form's custom `getresults()` method in order to populate the list box and fill the controls with the data from the result cursor's first record. The code snippet looks like this:

```
else
    thisform.iH = m.liH
    if !thisform.getresults()
        return .f.
    endif
endif
```

If the `SQLEXEC()` function call in the `getresults()` method fails, the user is notified and the form isn't opened.

Before we get to explaining the `getresults()` method in detail, let's finish up with the layout of the form. As you've seen, when you create the edit form from the `hwfrmnav` class, the list box and Done button come along for the ride, as do a pair of developer-only primary key fields.

Place the various controls on the form as shown in Figure 16. The LostFocus() method of each data entry control calls the custom validate_data() method. This routine detects differences between the values in the controls and the underlying database records and enables the Save button if there are any differences. This is the second quick and dirty decision – the controls don't do any additional validation, such as requiring a zip code to be of a certain format or ensuring that the street direction is limited to the values enumerated in the MySQL database (E/N/S/W or empty.)

Displaying data on the form

Back to getresults(). It JOINS the BIZ and LOC tables, creating a denormalized cursor (csrRes) for every Location record in the database, duplicating the business information as necessary. The primary keys for the Business and Location records are also placed in this cursor. Then the array that supports the results list box (hwlstnavres.aItems) is populated, and the highlight is placed on the first row in the list box.

```
* ui_edit.getresults()
debugout this.Name+'.getresults'

text to m.lcStr noshow
select biz.iidbiz, biz.cnabiz, biz.cnasec,
       loc.iidloc, loc.cno, loc.edir, loc.cstreet, loc.csuf, loc.csecline,
       loc.ccity, loc.cstate, loc.czip,
       iishq, tclosed
from BIZ, LOC
where BIZ.iidbiz = LOC.iidbiz
order by biz.cNaBiz, loc.cStreet
endtext
m.liX = sqlexec(thisform.iH, m.lcStr, "csrRes")
if m.liX > 0
    * display
    if used("csrRes") and reccount("csrRes") > 0
        m.liNumRows = reccount("csrRes")
        * fill array populating the listbox
        dimension thisform.hwlstnavRes.aItems[m.liNumRows,2]
        m.li = 1
        scan
            thisform.hwlstnavRes.aItems[m.li,1] = csrRes.cNaBiz
            thisform.hwlstnavRes.aItems[m.li,2] = csrRes.iidBiz
            m.li = m.li+1
        endscan
    else
```

```

    * display nothing
    dimension thisform.hwlstnavRes.aItems[1]
    thisform.hwlstnavRes.aItems[1] = [No Results.]
endif
else
    local aOps[1]
    m.liHowManyErrors = aerror(aOps)
    for m.li = 1 to m.liHowManyErrors
        insert into ZOOPS ;
        (inoerr, ctext, ctextodbc, csqldata, inoerrsql, ihandle, tadded) ;
        values ;
        (aOps[m.li,1], aOps[m.li,2], aOps[m.li,3], aOps[m.li,4], ;
        aOps[m.li,5], aOps[m.li,6], datetime())
    next
    * display nothing
    messagebox("SQL Execution failed; code is" +chr(10) +chr(13) +chr(10)
    +chr(13) ;
        +transform(m.lcStr))
    return .f.
endif
thisform.hwlstnavRes.ListIndex = 1

```

Highlighting the first row in the list box causes the controls to be populated with data. How? The list box is based on the hwlstnav class, which contains the anychange() method. anychange() presumes the existence of a form-level method, tabletoform(). Highlighting the first row calls the anychange() method, which in turn calls tabletoform(). And tabletoform() moves data from the array into the controls on the right side of the form. tabletoform() also populates the developer text boxes for the primary keys for the BIZ and LOC tables.

```

* ui_edit.tabletoform()
lparameters m.tiCurRow
debugout this.Name+'.tabletoform'
* if tiCurRow = 0, the cursor is empty
* the array has been filled
* we are redisplaying the form with a new row highlighted
* and new values in the individual controls that we get from the local cursor
* we may have to go back to the database for minor entity data
thisform.hwtxtdevIID1.Value = csrRes.iidBiz
thisform.hwtxtdevIID2.Value = csrRes.iidLoc
thisform.hwtxtBusiness.Value = csrRes.cNaBiz
thisform.hwtxtNaSec.Value = csrRes.cNaSec

```



```

thisform.hwtxtNo.Value = csrRes.cNo
thisform.hwtxtDir.Value = csrRes.eDir
thisform.hwtxtStreet.Value = csrRes.cStreet
thisform.hwtxtSuf.Value = csrRes.cSuf
thisform.hwtxtSecLine.Value = csrRes.cSecLine
thisform.hwtxtCity.Value = csrRes.cCity
thisform.hwtxtState.Value = csrRes.cState
thisform.hwtxtZip.Value = csrRes.cZip
thisform.hwchkHeadquarters.Value = csrRes.iishq
thisform.hwtxtClosed.Value = ttod(csrRes.tClosed)

```

Saving data

Obviously the Save button calls the save method, but before we get to the action in Save(), note that the anychange() method of the list box also calls Save() if the Save button is enabled. This is the third quick and dirty decision – if the user makes a change to a value in one record and then navigates to another record in the list box without explicitly saving the record, we just go ahead and save the data for them. Some folks would likely want to ask the user if they want to save before allowing them to move to another record, but that extra work kind of gets in our way for the time being.

The custom Save() is where the action is today. First, good values will be created from the data the user has entered. For example, each character string is passed through the z_es() function to escape quotation marks, so the user can save a value like

```
Bob's Extraordinary Pizza Parlor
```

Now the good stuff happens. Once we have valid data to stuff back into the MySQL database, we use the SQL UPDATE command to do the actual stuffing. It's time to ask ourselves, "Self, what record should we be updating?" In this case, we have to ask ourselves this twice, because we're updating both the BIZ and the LOC tables. Fortunately, we know what records in the back-end tables we're interested in because the primary keys for the records in question are in the developer-only text boxes. (When this application runs live, the text boxes are still available for the application to access – they're just not visible or enabled.)

We create an UPDATE command for the BIZ table, store it to m.lcStr, and execute it via SQLEXEC(). If the commit works, we store .t. to the m.llSaveWentWell variable, and .f. otherwise. Then we do the same UPDATE, m.lcStr, and SQLEXEC() things for the LOC table. This time, however, if the command executes successfully, we just leave the value of m.llSaveWentWell as is, since it's still true. If the second commit failed, store a .f. to the variable so we know we had at least one failure. Finally, alert the user if m.llSaveWentWell was set to .f. anywhere along the line and disable the Save button. (This all should be wrapped in a transaction, but one step at a time. We'll deal with transactions in the next chapter.)

Finally, the error reporting mechanism shown here could be substantially more robust, of course. An exercise left to the reader, I suspect.

```

* ui_edit.save()
debugout this.Name + '.save'
m.liidBiz = thisform.hwtxtdeviid1.value
m.liidLoc = thisform.hwtxtdeviid2.value
m.lcNaBiz = thisform.ocslib.z_es(alltrim(thisform.hwtxtBusiness.Value))
m.lcNaSec = thisform.ocslib.z_es(alltrim(thisform.hwtxtNaSec.Value))
m.lcNo = thisform.ocslib.z_es(alltrim(thisform.hwtxtNo.Value))
m.leDir = thisform.ocslib.z_es(alltrim(thisform.hwtxtDir.Value))
m.lcStreet = thisform.ocslib.z_es(alltrim(thisform.hwtxtStreet.Value))
m.lcSuf = thisform.ocslib.z_es(alltrim(thisform.hwtxtSuf.Value))
m.lcSecLine = thisform.ocslib.z_es(alltrim(thisform.hwtxtSecLine.Value))
m.lcCity = thisform.ocslib.z_es(alltrim(thisform.hwtxtCity.Value))
m.lcState = thisform.ocslib.z_es(alltrim(thisform.hwtxtState.Value))
m.lcZip = thisform.ocslib.z_es(alltrim(thisform.hwtxtZip.Value))
m.liisHQ = thisform.hwchkHeadquarters.Value
m.lcClosed = thisform.ocslib.z_tfed(thisform.hwtxtClosed.Value)

```

```

text to m.lcStr textmerge noshow

```

```

update BIZ

```

```

    set

```

```

    cNaBiz = '<<m.lcNaBiz>>',

```

```

    cNaSec = '<<m.lcNaSec>>',

```

```

    cchanged = 'bob',

```

```

    tchanged = now()

```

```

    where biz.iidbiz = <<m.liidBiz>>

```

```

endtext

```

```

m.liX = sqlexec(thisform.iH, m.lcStr)

```

```

if m.liX > 0

```

```

    * continue on

```

```

    m.llSaveWentWell = .t.

```

```

else

```

```

    local aOps[1]

```

```

    m.liHowManyErrors=AERROR(aOps)

```

```

    =thisform.ocslib.z_sqlerror(@aOps, m.lcStr, m.liHowManyErrors)

```

```

    m.llSaveWentWell = .f.

```

```

endif

```

```

text to m.lcStr textmerge noshow

```

```

update LOC set

```

```

cNo = '<<m.lcNo>>',
eDir = '<<m.leDir>>',
cStreet = '<<m.lcStreet>>',
cSuf = '<<m.lcSuf>>',
cSecLine = '<<m.lcSecLine>>',
cCity = '<<m.lcCity>>',
cState = '<<m.lcState>>',
cZip = '<<m.lcZip>>',
iishq = <<m.liisHQ>>,
tClosed = '<<m.lcClosed>>',
cchanged = 'bob',
tchanged = now()
where loc.iidloc = <<m.liidloc>>
endtext
m.liX = sqlexec(thisform.iH, m.lcStr)
if m.liX > 0
    * continue on
    * don't need to update savewentwell
    * if previous save was good, so was this, so still .t.
    * if previous save failed, we're going to return a .f.
    * regardless if this one worked
else
    local aOps[1]
    m.liHowManyErrors=AERROR(aOps)
    =thisform.ocslib.z_sqlerror(@aOps, m.lcStr, m.liHowManyErrors)
    m.llSaveWentWell = .f.
endif

if m.llSaveWentWell
    thisform.hwlblSaveFailed.caption = ""
else
    thisform.hwlblSaveFailed.caption = "An error has happened during save.
Please see the error log."
endif
thisform.hwcmdSave.Enabled = .f.

return m.llSaveWentWell

```

Once the data is saved, you may notice that, in the event of a business name change, the list box isn't updated afterwards. Again, you might want to try your hand at making this happen. If not, never fear; we add this in shortly, in a future example.

As before, the Close() method, called from the Done button, closes the cursor and disconnects from the database. And that's all there is to our first editing form.

An alternative to using the “developer-only” text boxes is setting up properties in the form to represent the primary key values; I like the text boxes because it enables easier debugging for those times when a typo results in the wrong primary key being stuffed into a table.

It's now time for you to open up the source for this example and play around with it, adding a list box refresh, perhaps, or changing the Street Direction text box to a combo box that limits the choices available to the user to those allowed by the database's field definition. Once you're comfortable, you might consider changing the interface to include separate buttons for saving the business data versus the location data, or maybe even reworking the interface so it's not denormalized as it is here.

A simple two table delete form

(*ui_delete.scx*)

By now you're getting the hang of it. Once you make a connection and haul the data into your form, you can work with it via a local cursor just as you would local tables. When you're ready to commit your changes, you fire back to the database and you're done. Our delete form is just like the ui_edit form, except with a new Delete button and the corresponding Delete() method. We'll also add the call to refresh the list box afterwards because it wouldn't do to allow the user to try to delete the same record twice. **Figure 17** shows the finished ui_delete form.

Simple Edit and Delete (Handle: 1)

Antonio's Upper Crust Pizza
Any Which Way But Loose
Armin Koch Furniture Co Inc
Artique
Artisan Wall Covering
Associated Bank
Au Courant Inc
Bailey Law Office
Balia Wellness Center
Barnett, Nancy A DDS
Bay Bakery
Bay Shore Lutheran
Bay Travel
Benz Cyclery
Betsy & Ken Head, Realty Exec
Betty Johnson Interiors
Billy Klingman, Realty Exec Lakeshore
Bluefish Media Design
Body Therapy Assoc
Boves, Peter J
Breadsmith Bakery

☒ Headquarters

Business: Any Which Way But Loose

Line 2: Clyde's our speciality

Street: 4 West Chester Main

Line 2:

City: Los Alamos

State: NM Zip: 84501

Out of Business: ...

Delete Save Done

idbiz 200 idloc 188

Figure 17. A simple two table delete form.

Denormalized table issues

We simply (don't you get a chill down your spine when you hear 'simply'?) call the custom Delete() method when the user hits the Delete button. Inside that method, though, there are some decisions to be made based on this parent-child interface. Since the list box is denormalized, the list can contain two types of rows. The first type of row contains a name of a business that has only one location; the other type of row contains a name of a business that has multiple locations (although the row in question represents just one of those locations.)

If the highlighted row in the list is the first type, the work to perform when the user clicks the Delete button is easy – delete both the location and the business it belongs to. The second, though, is trickier.

Do we want to delete the location but leave the business record intact, since there are other children (locations) attached to the business? Probably. After all, we should be aware that the reason the user is deleting a record is most likely because the location closed, fell into a sinkhole, or was bought by someone else. (The user might also be deleting the record because it was entered in error.) The bottom line is that this specific location is no longer affiliated with the business.

What if, however, the entire business was shut down? In this scenario, it would be convenient to give the user the option to delete all locations for this business, and then delete the business itself. It may be the case that you decide on this rule as an absolute for the entire system, and structure the database that way – business has a one-to-one or -more relationship to location.

MySQL, like VFP, has the ability to set up referential integrity on the database server, so all of the locations (children) are automatically deleted when the business (parent) is deleted, or that the parent business is deleted when no child locations are left. In this example, we'll do it manually.

First check if there is only one location, and if so, delete the biz and the location. If there is more than one location, we just delete the location and leave the business alone.

This form is an enhancement of the ui_edit form, which means that we call this form the same way:

```
do form ui_delete with 'localhost', 'whil', 'secret'
```

passing the parameters that are handled in the Init() of the form.

The delete() method

Under the hood, this form is virtually identical to the ui_edit.scx form, so if you're building this yourself, create the form by making a copy of ui_edit.scx, calling the result ui_delete.scx.

The only differences between this and the ui_edit form are the Delete button and the Delete() method. The Delete button's Click() method calls the form's Delete() method.

The Delete() method grabs the primary keys for the BIZ and LOC tables, and then stores the current ListIndex value of the list box, so we can return the highlight to a logical place after deleting the current row. Then the ListIndex value is tested to make sure a row in the list box is highlighted, so we know which row we want to delete.

As soon as the user clicks the Delete button, we ask the user to confirm their intention for deletion, count how many locations belong to this record's Business, and then delete the location. Then, if there's just this one location for the business, delete the business too. We set a flag upon successful deletion of the location so we don't accidentally delete a business after being unable to delete the child location. We also don't delete the business if this location has sibling locations.

Finally, we call the form's custom `getresults()` method to repopulate the list box, and set the highlight to the same row in the list box, which means the row after the row being deleted becomes the new highlighted row. (In order to keep things simple, this example doesn't handle the case where you were already on the last row of the list, and then deleted the last row. In such a situation, you would want to check to see if `this.ListIndex = this.ListCount`, and if so, set the `ListIndex = m.liOriginalListIndex - 1`.)

```
* ui_delete.delete()
debugout thisform.Name + '.delete'

m.liidBiz = thisform.hwtxtdeviid1.value
m.liidLoc = thisform.hwtxtdeviid2.value
m.liOriginalListIndex = thisform.hwlstnavres.ListIndex

* make sure on a record
* determine if > 1 loc
* get name of biz/loc
* ask user to confirm
* delete
* refresh list box
* reposition to previous record

if thisform.hwlstnavres.ListIndex = 0
    return
endif

m.lcToDelete = thisform.hwtxtBusiness
if messagebox("Are you sure you want to delete " + chr(13) + chr(10) ;
    + thisform.hwtxtBusiness.Value + chr(13) + chr(10) ;
    + " on " + thisform.hwtxtStreet.Value + thisform.hwtxtSuf.Value + chr(13) +
chr(10) ;
    + "forever?",4+256) <> 6 && not yes (no)
    return .f.
endif

text to m.lcStr textmerge noshow
```

```

select count(iidloc) as cHowMany from LOC
  where loc.iidbiz = <<m.liidBiz>>
endtext
m.liX = sqlexec(thisform.iH, m.lcStr, "csrCount")
if m.liX > 0
  * continue on
  if val(csrCount.cHowMany) > 1
    m.llThereIsJustOne = .f.
  else
    m.llThereIsJustOne = .t.
  endif
else
  local aOps[1]
  m.liHowManyErrors=AERROR(aOps)
  =thisform.ocslib.z_sqlerror(@aOps, m.lcStr, m.liHowManyErrors)
  * we don't know how many others there are, so assume there is more than one
  m.llThereIsJustOne = .f.
endif

* delete the loc regardless
m.llDeleteLocWasGood = .f.
text to m.lcStr textmerge noshow
  delete from LOC where loc.iidloc = <<m.liidLoc>>
endtext
m.liX = sqlexec(thisform.iH, m.lcStr)
if m.liX > 0
  * continue on
  m.llDeleteLocWasGood = .t.
else
  local aOps[1]
  m.liHowManyErrors=AERROR(aOps)
  =thisform.ocslib.z_sqlerror(@aOps, m.lcStr, m.liHowManyErrors)
  * couldn't delete the LOC
  m.llDeleteLocWasGood = .f.
endif

m.llDeleteBizWasGood = .f.
if m.llThereIsJustOne and m.llDeleteLocWasGood
  debugout "there is just one and del loc was good so deleting biz"

```

```

* delete the biz too
text to m.lcStr textmerge noshow
  delete from BIZ where biz.iidbiz = <<m.liidBiz>>
endtext
m.liX = sqlexec(thisform.iH, m.lcStr)
if m.liX > 0
  * continue on
  m.llDeleteBizWasGood = .t.
else
  local aOps[1]
  m.liHowManyErrors=AERROR(aOps)
  =thisform.ocslib.z_sqlerror(@aOps, m.lcStr, m.liHowManyErrors)
  * couldn't delete the LOC
  m.llDeleteBizWasGood = .f.
endif
else
  * don't delete the biz cuz
  * - there are other, or
  * - the Loc delete wasn't successful
endif

* refresh list box
thisform.getresults()

* reposition if m.llDeleteLocWasGood
thisform.hwlstnavres.ListIndex = m.liOriginalListIndex
thisform.hwlstnavRes.SetFocus()

if used("csrCount")
  use in csrCount
endif

```

There are two schools of thought with respect to asking the user to confirm a deletion. The first is not to ask for confirmation; don't insult the user's intelligence – they clicked the button, assume they know what they're doing. The other school, though, is that they may have inadvertently clicked the Delete button when they meant to click somewhere else nearby. With MySQL, not allowing the user to recover from such a mistake via a confirmation dialog is doubly important, because there is no 'undo' functionality in MySQL.

Specifically, a delete in MySQL (as with other SQL databases) is not like a 'delete' in VFP. 'Delete' in MySQL is like 'delete' followed by 'pack' in VFP. There is no 'delete flag' in the

MySQL database architecture. As a result, it's critical that you make sure to have the user confirm the pending deletion with an "Are you sure?" dialog.

Some folks find it handy to use an alternative method, simulating VFP's delete flag, by using a "iIsDeleted" flag in each table, setting that flag to 1 to indicate the record is deleted, and when doing SQL SELECTS, making sure to include the "where iIsDeleted = 0" clause all the time. This will only work, however, if you're not using MySQL's Referential Integrity mechanisms.

What else might you want to do in your spare time? Alerting the user that there are multiple locations of the business and asking them if they want to delete all of the locations is one possibility. Or you could notify the user if they are deleting the HQ location, so they can assign another location as the HQ if they so wish. You might also try allowing the user to click on multiple records in the list box, and then delete all that are marked with a single click of the delete button. Again, you want to make sure you require the user to confirm their intent.

In a business sense, note that deletion may not be what some users want to do; rather, they'll prefer to mark a location 'closed' by entering a Closed date. But we needed to learn how to delete records, right?

An all-in-one form

(ui_allinone.scx)

Each of the previous forms works pretty well, but real life is rarely that simple. Now it's time to put add, edit, and delete capabilities into one form, together with a database schema that includes parents, children, and grandchildren.

Design of an all-in-one form

As anyone who has attempted to put together a 'name and address' database knows, there are so many combinations and permutations of the designs that it's nearly impossible to do justice to every scenario. As a result, you'll have to live with the tacit assumptions I've made in designing the database and assembling the interface for this particular application.

This "all-in-one" form starts out life as the ui_delete form we just looked at, but has a number of extensions added to it. First, we don't dump the entire BIZ/LOC join cursor into the list box; instead, we allow the user to filter the list box by the first letter. Within that first letter, we display all business/location combinations like in previous examples, so it's possible to see several "Starbucks" rows in the list box, albeit each row representing a different location. On second thought, given the proliferation of coffee shops, seeing several Starbucks is pretty much inevitable.

We also allow the user to choose if they want the first letter to filter the business name, or the street name, so the user can view all of the businesses that begin with 'Q', or all of the businesses located on streets that begin with the letter 'S'. In the former case, the list is sorted by business name, and then by street under each business. In the latter case, the list is sorted by street name, then by direction, and then by street number, so that 76 East Silver Spring Drive will show up before 107 West Silver Spring Drive, and both of those will appear before 43 North Snowshoe Circle.

While this interface wouldn't do for databases with tens or hundreds of thousands of records, the small sample tables included here will work fine.

This form brings back the BizType child and Coordinate and Person grandchild entities seen earlier. The Business Type list box, populated from the BIZCAT table (which does a lookup into the ZLOOKUP table), contains one or more records that identify the type of business (restaurant, grocery, physicians office, etc.)

The tables that support the Contacts and Folks list boxes are both related to the LOC table. "Contacts" are what some people call 'coordinates' – telephone numbers, email addresses, Web URLs, and so on. Each location likely has its own voice and fax numbers, and may well have its own Web site. Others, however, may have a single Web site for all locations.

It is possible to create a design where some coordinates are attached the to Biz record (like the Web address or a main phone number) while other coordinates stay with a specific location (such as the fax machine at a specific store). However, this gets more complicated than what I want to deal with in this example. As a result, those coordinates that normally would be attached directly to the business are attached to the location identified as the headquarters for the business.

"Folks" are the people who belong to the business; again, they most likely would be attached to a specific location, although some folks who roam between multiple locations might be attached to the headquarters location for practicality.

You'll find "add (+)" and "delete (–)" buttons to the right of each set of controls for a table. The Add buttons open a new dialog that allows the user to add one or more records to that entity; to wit, the Add button to the right of the Business text boxes opens the Add dialog that we started out with in this chapter, while the Add button to the right of the Folks list box allows the user to add one or more people to the current location, using the same "Save and Close" and "Save and Add Another" mechanisms.

Edits to the Business and Location controls in the list box enable and are saved by the "Save" button at the bottom of the form; changes to the other controls are made only by adding and deleting entire records.

The entire form is shown in **Figure 18**.

The screenshot shows a software window titled "All-In-One (Handle: 1)". At the top, there is a row of letters from A to Z, with 'R' highlighted. Below this, a "Sort list by" section has radio buttons for "Company Name" (selected) and "Street Name".

The main area is divided into two panes. The left pane contains a list of businesses with their street addresses right-aligned:

Ralph's Fine Automobiles	
Ray's Florsit And Gift Baskets	6246 N Port Washington
Re/Max Lakeside	1200 E Capitol
Rossman Physical Therapy	3970 N Oakland Ave
Roundy Memorial Baptist	1250 E Hampton
Royal Garden	206 W Silver Spring
Royal Palace	
Ruhama's Yarn And Needlepoint	420 E Silver Spring

The right pane contains form fields for the selected business, "Rossman Physical Therapy":

- Business:** Rossman Physical Therapy
- Line 2:** (empty)
- Type of Biz:** Health, Misc
- Headquarters:** ☒ (checked)
- Street:** 3970 N Oakland Ave
- Line 2:** Suite 703
- City:** Shorewood
- State:** WI
- Zip:** 53211
- Out of Business:** ☐
- Contacts:** voice 414.963.6330
- Folks:** Paul D. Rossman P.T. S.C.

At the bottom, there are "Save" and "Done" buttons, and a status bar showing "186" in two yellow boxes.

Figure 18. The "all-in-one" form provides add/edit/delete capabilities for multiple related tables.

A number of niceties have been added to this form. For example, each of the letters at the top of the Business/Location list box act as a filter control, much like some Web sites use. After clicking a letter at the top of the Business/Location list box, that letter is disabled. Figure 18 shows the letter 'R' disabled after clicking it displays all of the businesses that begin with the letter 'R'.

The street addresses of businesses are displayed in the list box to allow the user to distinguish between multiple locations of the same business. I considered using two list boxes; one solely for business names and a second to display locations, but decided against it after seeing the capability to filter and sort the list by street name was more important than the 'purist' approach of normalizing the business name and street address combination.

If a business doesn't have a location attached, no street address displays in the left-hand list box. "Ralph's Fine Automobiles" and "Royal Palace" are two such examples.

You'll see that the street numbers are right justified, making it easier to distinguish between, say, "57 West Fifth" and "511 West Fifth." It also aids in sorting the list; the user can see the street numbers progress from smallest to largest. See **Figure 19**.

The screenshot shows a software application window titled "All-In-One (Handle: 1)". It has a menu bar with letters A-Z and a toolbar with "Save" and "Done" buttons. The main area is divided into two panes. The left pane contains a list of businesses with their names and addresses, sorted by "Company Name". The right pane shows details for the selected business, "Clinic Of Cosmetic Surgery", including its address, phone number, and a list of contacts. The "Folks" section shows "No Results."

Figure 19. Street numbers are right justified.

List boxes display the text “No Results” if there are no records for that entity.

Internal design

Let’s take a look under the hood.

This form is yet another enhancement to the `ui_edit` form, so it’s based on the `hwfrmnav` class in `hwctrl.vcx`. This means it’s called like so:

```
do form ui_allinone with 'localhost', 'whil', 'secret'
```

The form has one new property, `cWhichLetter`, that identifies which filter letter is currently active. There are a number of new methods, all dealing with the deletion of records; I’ll cover those shortly.

Much of the same code from `ui_edit` is still found in `ui_allinone`, so I’ll just point out the differences. In the `Init()`, the new form property, `cWhichLetter`, is assigned its initial value, ‘A’, and the ‘A’ label is initially disabled.

```
* ui_allinone.init() (partial)
if m.liH < 1
    messagebox("No connection; handle is:"+transform(m.liH)+":")
    return .f.
else
    thisform.iH = m.liH
    thisform.Caption = alltrim(thisform.Caption) + " (Handle: " +
transform(thisform.iH) + ")"
    thisform.hwlblSaveFailed.Caption = ''
    thisform.cWhichLetter = 'A'
```

```

thisform.hwlblpickA.ForeColor = RGB(0,0,0)
thisform.hwlblpickA.FontUnderline = .f.
if !thisform.getresults()
    return .f.
endif
endif
endif

```

The `getresults()` method has been modified to handle the letter and the 'Business or Street' filters. The letter filter is assigned to a variable, like so:

```

if thisform.hwopgCompanyOrStreet.Value = "Company Name"
    m.lcFilter = "cNaBiz like '" + thisform.cWhichLetter + "%'"
else
    m.lcFilter = "cStreet like '" + thisform.cWhichLetter + "%'"

```

Choosing which WHERE clause to use is based on the value of the Business or Street option group.

```

* ui_allinone.getresults() (partial)
text to m.lcStr textmerge noshow
select biz.iidbiz, biz.cnabiz, biz.cnasec,
    loc.iidloc, loc.cno, loc.edir, loc.cstreet, loc.csuf, loc.csecline,
    loc.ccity, loc.cstate, loc.czip,
    iishq, tclosed
from BIZ left join LOC
on (LOC.iidbiz = BIZ.iidbiz)
where <<m.lcFilter>>
order by biz.cNaBiz
endtext

```

The `tabletoform()` method that moves data from the `csrRes` cursor created by `getresults()` has been modified to also query the BIZCAT, COOR, and PERSON tables for the records related to the chosen Business and Location.

```

* ui_allinone.tabletoform() (partial - just fill one listbox)
* coordinates listbox
text to m.lcStr noshow
select iidcoor, cdata, etype from COOR
    where coor.iidloc = ?thisform.hwtxtdevIID2.value
endtext
m.liX = sqlexec(thisform.iH, m.lcStr, "csrResCoor")
if m.liX > 0
    * display

```

```

if used("csrResCoor") and reccount("csrResCoor") > 0
    m.liNumRows = reccount("csrResCoor")
    * fill array populating the listbox
    dimension thisform.hwlstCoor.aItems[m.liNumRows,2]
    m.li = 1
    scan
        thisform.hwlstCoor.aItems[m.li,1] = csrResCoor.eType + " " +
csrResCoor.cData
        thisform.hwlstCoor.aItems[m.li,2] = csrResCoor.iidCoor
        m.li = m.li+1
    endscan
else
    * display nothing
    dimension thisform.hwlstCoor.aItems[1]
    thisform.hwlstCoor.aItems[1] = [No Results.]
endif
else
    local aOps[1]
    m.liHowManyErrors = aerror(aOps)
    for m.li = 1 to m.liHowManyErrors
        insert into ZOOPS ;
        (inoerr, ctext, ctextodbc, csqldata, inoerrsql, ihandle, tadded) ;
        values ;
        (aOps[m.li,1], aOps[m.li,2], aOps[m.li,3], aOps[m.li,4], ;
        aOps[m.li,5], aOps[m.li,6], datetime())
    next
    * display nothing
    dimension thisform.hwlstCoor.aItems[1,2]
    thisform.hwlstCoor.aItems[1,1] = [Bad Results.]
    thisform.hwlstCoor.aItems[1,2] = 0
endif
thisform.hwlstCoor.Requery()
thisform.hwlstCoor.ListIndex = 1

```

Adding and deleting minor entities

The Add buttons each contain a couple lines of code in the Click() method. This code calls a separate Add form for that entity. However, the Delete buttons each call a separate delete method in their Click() method. Why the difference? Well, because I'm lazy. Strictly speaking, I should have created separate add methods that were called from the Add button clicks. Code that does 'work' doesn't belong in the Click() of a command button. The function of a command button, as

Andy Kramek so eloquently puts it, is to notify some object that an action is needed. If the add code got any longer or more complex, I'd move it into separate methods where it belongs.

Adding a Business

Clicking the Add Business button brings forth the same form shown in Figure 1. However, the actual code that calls the form is slightly different. This is in the Click() method of the button:

```
* ui_allinone.hwcmdaddbiz.click()
do form ui_allinone_addbiz with thisform.iH, thisform.hwtxtdeviid1.value,
    thisform.ocslib to m.liidnew
if m.liidnew > 0
    thisform.getresults()
else
    * no biz added
endif
```

As you can see, the Add Business form is called with parameters that reference the calling form's connection handle and the ocslib object. (The reference to the primary key isn't necessary in this specific form but will be used in other "Add" forms. I kept it in there in case I wanted to abstract the whole function.) Doing so enables the Add Business form to communicate with the database server using the same connection, and have access to the client-server functions created in the Init() of the calling form.

Inside the Add Business form, everything functions pretty much like it did in the stand-alone example earlier in this chapter, with one important exception. The Save() method, after successfully inserting the new business record, does a second call to the database, requesting the primary key of the record just added, like so:

```
* ui_allinone_addbiz.save()
if m.liX > 0
    * insert was successful
    text to m.lcStr noshow
        select last_insert_id() as cLastID
    endtext
    m.liX = sqlexec(thisform.iH, m.lcStr, "csrLastID")
    if m.liX > 0
        thisform.iLastID = val(csrLastID.cLastID)
    else
        thisform.iLastID = 0
    endif
```

This value, iLastID, is returned by the Add Business form to the calling form (in the Unload method). The calling form, ui_allinone, now knows the last record added to the BIZ table, and

this value can be used to navigate to that record and reposition the interface on that record.

Deleting a Business

The Delete Business button calls the deletebiz() method, which contains pretty much the same code as in the ui_delete form. The only difference is that instead of just deleting the location attached to the business (if there's only one location) the method also deletes all of the child records – Business Types attached to the Business as well as Contacts and Folks attached to the single location.

```
* ui_allinone.deletebiz()
debugout thisform.name + '.deletebiz'

m.liidBiz = thisform.hwtxtdeviid1.value
m.liidLoc = thisform.hwtxtdeviid2.value
m.liOriginalListIndex = thisform.hwlstnavres.ListIndex

* make sure on a record
if thisform.hwlstnavres.ListIndex = 0
    return
endif

* force confirm
if messagebox("Are you sure you want to delete the business " + chr(13) +
chr(10) ;
    + thisform.hwtxtBusiness.Value + chr(13) + chr(10) ;
    + "forever?",4+256) <> 6 && not yes (no)
    return .f.
endif

* determine how many loc
m.llThereIsAtMostOne = .t.
text to m.lcStr textmerge noshow
    select count(iidloc) as cHowMany from LOC
    where loc.iidbiz = <<m.liidBiz>>
endtext
m.liX = sqlexec(thisform.iH, m.lcStr, "csrCount")
if m.liX > 0
    * continue on
    m.liHowManyLoc = val(csrCount.cHowMany)
    * note that this ID value is valid only if there was one loc for the biz
```



```

else
    local aOops[1]
    m.liHowManyErrors=AERROR(aOops)
    =thisform.ocslib.z_sqlerror(@aOops, m.lcStr, m.liHowManyErrors)
    * we don't know how many others there are, so assume there is more than one
    m.liHowManyLoc = -1
    m.liidLoc = 0
endif

* delete
* will delete if 0 or 1 locations
* will warn and not allow if > 1 location
do case
case m.liHowManyLoc < 0
    messagebox("Unable to delete business because count of locations failed.")
    return .f.

case m.liHowManyLoc = 0
    * no loc to delete, just going to delete the biz
    * delete the biz types
    thisform.deletebizcat()

case m.liHowManyLoc = 1
    * going to delete the coordinates
    thisform.deletecoor()
    * delete the persons
    thisform.deleteperson()
    * delete the sole loc
    thisform.deleteloc()
    * delete the biz types
    thisform.deletebizcat()

otherwise
    * lots of locations, so we don't delete locations
    * AND we don't delete business

    messagebox("There is more than one location attached to this business.
Deleting the business would leave the rest of the locations without a business
to belong to. Delete cancelled. Delete the other locations first, if you truly
want to delete the business.")

    return .f.

```

```

endcase

* now delete the biz
m.llDeleteBizWasGood = .f.
text to m.lcStr textmerge noshow
    delete from BIZ where biz.iidbiz = <<m.liidBiz>>
endtext
m.liX = sqlexec(thisform.iH, m.lcStr)
if m.liX > 0
    * continue on
    m.llDeleteBizWasGood = .t.
else
    local aOops[1]
    m.liHowManyErrors=AERROR(aOops)
    =thisform.ocslib.z_sqlerror(@aOops, m.lcStr, m.liHowManyErrors)
    * couldn't delete the LOC
    m.llDeleteBizWasGood = .f.
endif

* refresh list box
thisform.getresults()

* reposition
thisform.hwlstnavres.ListIndex = m.liOriginalListIndex
thisform.hwlstnavRes.SetFocus()

if used("csrCount")
    use in csrCount
endif

```

Adding and deleting Business Types

The Add Business Type form, called from the Add button next to the Type of Biz field, is shown in **Figure 20**.

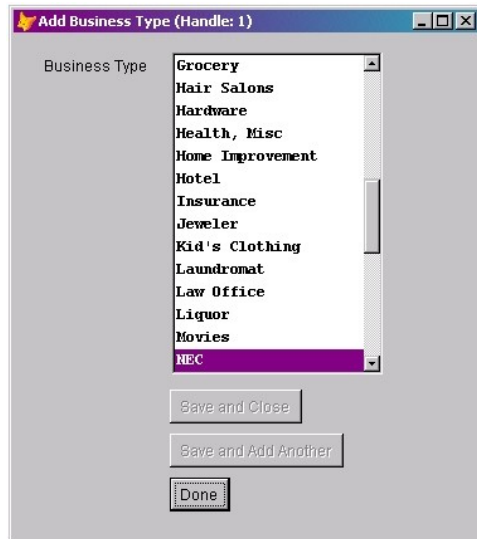


Figure 20. The Add Business Type form allows the user to add one or more Business Types to a Business.

To understand what’s happening in this form, it’s important to revisit the structure of the BIZCAT table. The fields of interest with a couple of sample records are displayed here:

iidbizcat	iidbiz	iidcat
1	10	24
2	10	57

The first field is a surrogate primary key for the BIZCAT table. The second field points to the primary key for the BIZ table, tying this Business Type to a specific Business record. The third field points to the primary key for the ZLOOKUP table – identifying the Business Type record in the lookup table. Thus, for iidbizcat = 1, iidbiz = 10, pointing to record 10 in BIZ (such as “Benji’s Deli”), while iidzlookup points to record 24 (say, “Bakery”). The second record is also for Benji’s Deli, and the iidzlookup record, # 57, points to “Restaurant.”

Unlike the Add Business form, the “hwtxtdeviid1.value” parameter passed to this form is important, because the value holds the primary key of the current business record. As a result, it becomes one of the foreign keys in the BIZCAT table for the new BIZCAT record. And what about the other foreign key? Hold your horses for just a moment.

The Add Business Type form also differs from the Add Business form in that it goes out to the database and queries the ZLOOKUP table for all valid CAT (Business Type) records in order to populate the list box. The default value in the list box, NEC, stands for “Not Elsewhere Classified” and serves as a default for Businesses whose owners don’t know what business they’re in yet. (Eric Selje correctly notes that, outside of Silicon Valley, this is considered to be a bad business plan.)

```
* ui_allinone_addbiztype.init()
lparameters m.tiH, m.tiIDBiz, m.toCS
debugout this.Name + '.Init'
```

```

thisform.iH = m.tiH
thisform.ocslib = m.toCS
thisform.iidparent = m.tiidbiz

dodefault()

thisform.Caption = alltrim(thisform.Caption) + " (Handle: " +
transform(thisform.iH) + ")"
thisform.hwlblSaveFailed.Caption = ''

* populate the Business Type cbo
m.liHowMany = 0
text to m.lcStr textmerge noshow
    select cde, iidzlookup from ins.zlookup
        where cnallookup = 'CAT'
        order by cde
endtext
m.liX = sqlexec(thisform.iH, m.lcStr, "csrResCat")
if m.liX > 0
    * display
    if used("csrResCat") and reccount("csrResCat") > 0
        m.liNumRows = reccount("csrResCat")
        * fill array populating the listbox
        dimension thisform.hwlstBizType.aItems[m.liNumRows,2]
        select csrResCat
        m.li = 1
        scan
            thisform.hwlstBizType.aItems[m.li,1] = csrResCat.cDe
            thisform.hwlstBizType.aItems[m.li,2] = csrResCat.iidzlookup
            m.li = m.li+1
        endscan
    else
        dimension thisform.hwlstBizType.aItems[1,2]
        thisform.hwlstBizType.aItems[1,1] = "No Results."
        thisform.hwlstBizType.aItems[1,2] = 0
    endif
else
    local aOps[1]

```

```

m.liHowManyErrors=AERROR(aOops)
=thisform.ocslib.z_sqlerror(@aOops, m.lcStr, m.liHowManyErrors)
dimension thisform.hwcboBizType.aItems[1,2]
thisform.hwcboBizType.aItems[1,1] = "No Results."
thisform.hwcboBizType.aItems[1,2] = 0
return .f.
endif
thisform.hwlstBizType.Requery()
thisform.hwlstBizType.ListIndex = 1

m.liRowNumber=ascan(thisform.hwlstBizType.aItems, "NEC", -1, -1, 1, 8)
thisform.hwlstBizType.ListIndex = m.liRowNumber
thisform.hwlstBizType.Refresh()

```

The second column in the array that supports the Business Type list box is populated with the primary key of the ZLOOKUP table for the appropriate Business Type records. This key becomes the foreign key pointing to the ZLOOKUP table – the second foreign key in the BIZCAT table.

As a result, the Save() method ends up just stuffing a couple of foreign keys into the BIZCAT table.

```

* ui_allinone_addbiztype.save() (partial)
* save data
debugout this.Name + '.save'
m.liidbiz = thisform.iidparent
m.liidcat = thisform.hwlstBizType.aitems[thisform.hwlstBizType.listindex,2]
text to m.lcStr textmerge noshow
    insert into INS.bizcat
        (iidbiz, iidcat, cadded, tadded, cchanged, tchanged)
        values
        (<<m.liidbiz >>, <<m.liidcat>>, 'bob', now(), 'bob', now() )
endtext
m.llSaveWentWell = .f.
m.liX = sqlexec(thisform.iH, m.lcStr)
if m.liX > 0
    * continue on
    text to m.lcStr noshow
        select last_insert_id() as cLastID
    endtext
    m.liX = sqlexec(thisform.iH, m.lcStr, "csrLastID")

```

```

if m.liX > 0
    thisform.ilastid = val(csrLastID.cLastID)
else
    thisform.ilastid = 0
endif
return .t.
else
    local aOps[1]
    m.liHowManyErrors=AERROR(aOps)
    =thisform.ocslib.z_sqlerror(@aOps, m.lcStr, m.liHowManyErrors)
    return .f.
endif

```

Adding and deleting Contacts

The Add Contact form is shown in **Figure 21**.

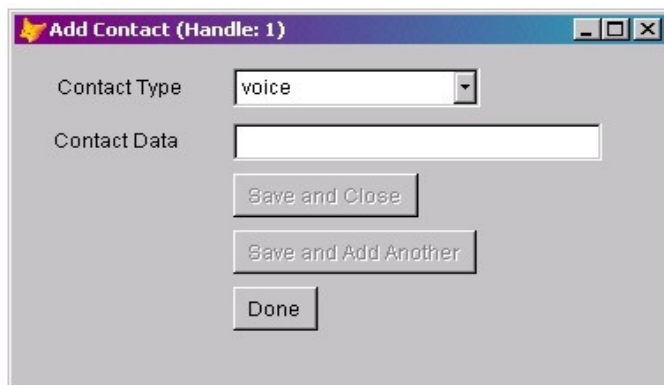


Figure 21. The Add Contact form lets the user add one or more coordinates, such as telephone numbers or email addresses, to a Location.

Analogous to the Add Business Type form, which is passed the `hwtxtdeviid1.value` parameter, the `hwtxtdeviid2.value` parameter passed to this form is used as the foreign key in the COOR table when a new contact record is added, because COOR is tied to LOC (whose primary key is found in iid2, compared to iid1 that contains the primary key for BIZ).

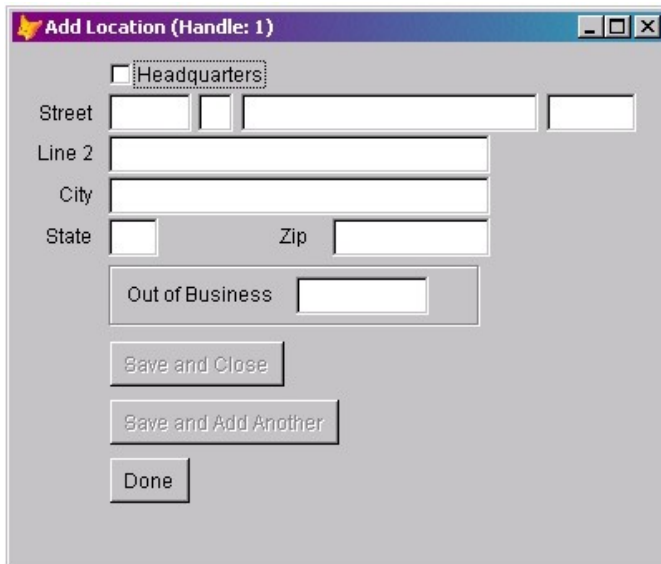
The Contact Type combo box is also populated via a lookup against the ZLOOKUP table, in order to provide friendly labels for the various contact types.

Unlike the Add Business Type form, though, the foreign key from the ZLOOKUP table for the Contact Type isn't stuffed into the new contact record; instead, the actual text value is inserted. The text for various Business Types is likely to change for at least some Business Types, so we put the primary key to the Business Type record in the BIZCAT table. On the other hand, the strings "voice" and "fax" aren't likely to change any time in the foreseeable future, and putting the actual data in the COOR table makes reporting and lookups just that much easier. Using a combo box instead of a text box where the user could enter free form data ensures

consistency in the values of Contact Type. Using a lookup table to populate the combo box ensures that we can add new contact types easily.

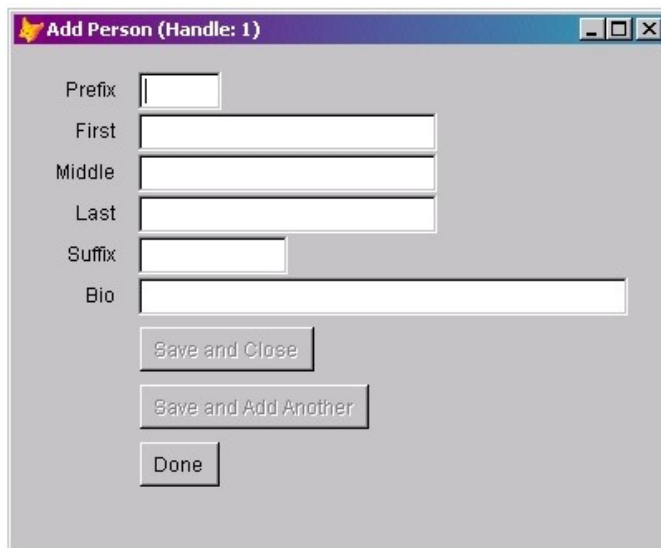
Thus, the save method for contact stuffs the FK to COOR and the text for Contact Type and Contact data into a new record in COOR.

The Add Location (**Figure 21**) and Add Person (**Figure 22**) forms function identically to the Add Business form, providing controls to enter the appropriate data.



The 'Add Location' dialog box has a title bar with a star icon and the text 'Add Location (Handle: 1)'. It contains a checkbox labeled 'Headquarters'. Below this are text input fields for 'Street', 'Line 2', 'City', 'State', and 'Zip'. There is also a group box labeled 'Out of Business' containing a text input field. At the bottom are three buttons: 'Save and Close', 'Save and Add Another', and 'Done'.

Figure 21. The Add Location dialog lets the user add a physical location to an existing business.



The 'Add Person' dialog box has a title bar with a star icon and the text 'Add Person (Handle: 1)'. It contains text input fields for 'Prefix', 'First', 'Middle', 'Last', 'Suffix', and 'Bio'. At the bottom are three buttons: 'Save and Close', 'Save and Add Another', and 'Done'.

Figure 22. The Add Person dialog lets the user add a contact person to a specific location.

Key Concepts

1. VFP + MySQL is the most inexpensive and powerful combination for C/S development on the planet
2. Configuring MySQL permissions
3. Only working with a subset of the backend at any one time

Copyright, 2007, Whil Hentzen.