Contents

Orders Example Database

The Orders example database is a REALbasic database application that manages customers, products, and orders. With the example and this guide, you'll learn how to use REALbasic's database classes in your projects and how to use object-oriented REALbasic features to rapidly build database-driven applications.

You will learn how to:

- Create a database and its tables and fields,
- Design a user interface,
- Add, modify, and delete records,
- Search for records,
- Import, export, and print records.

The Orders example database uses the built-in REAL SQL Database as its data source. However, most of the skills you will learn here do not apply only to REAL SQL Database. They also apply to all the data sources that REALbasic supports. This is because REALbasic abstracts all but one of the database classes that the developer uses from the data source. This means that you can write an application using the REAL SQL Database engine and can later change just a few lines of code that create the Database subclass instance to move the entire the project to another data source.

The example database that accompanies this Guide is written specifically for the single-user REAL SQL Database. It can easily be modified to support multi-user applications that use the optional REAL SQL Server database. The REAL SQL Server is a separate REAL Software product. It adds special commands for record locking and unlocking, a password access system with an extensive privileges architecture, an integrated backup system, a server Admin utility, and the ability to develop a customized server admin interface within REALbasic applications.

Creating the Database

In order to use a database in REALbasic, the database back-end must be created. The back-end contains the tables that store data, while the front-end is the user interface to the database.

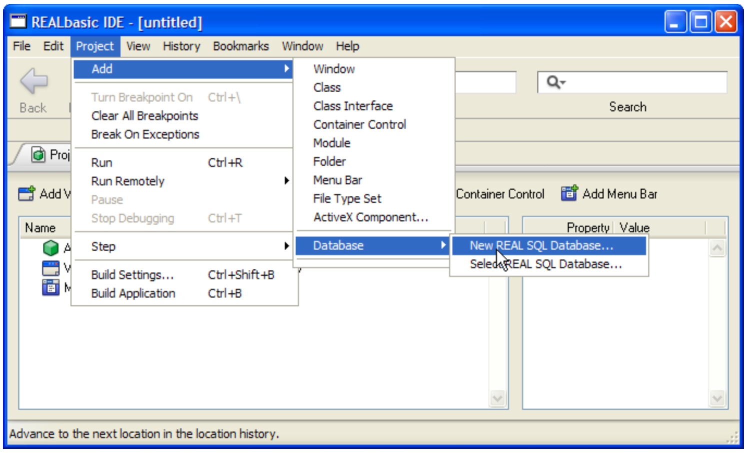
You can create the database in one of two ways. A REAL SQL Database can either be created in the REALbasic IDE at design time and added to the REALbasic project or it can be created dynamically (with code) and accessed at runtime.

This project uses the second alternative, but we will first describe the first alternative.

Using the IDE's Database Editor

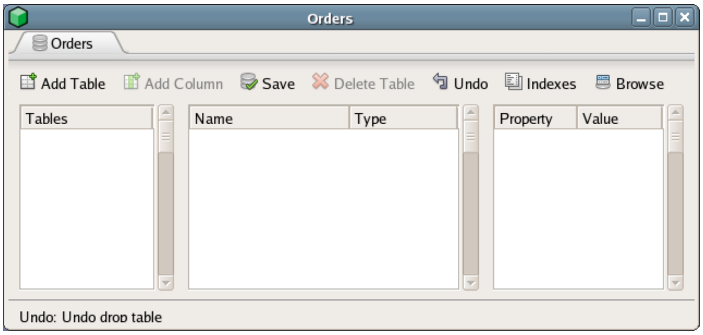
To create a new REAL SQL Database using the REALbasic IDE, choose Project ▶ Add ▶ Database ▶ New REAL SQL Database or right+click in the Project Window and use the contextual menu.

Figure 1. Adding a new REAL SQL Database source to a REALbasic project.



A save-file dialog appears, asking you to save the database somewhere on your computer. After saving the database file, REALbasic adds the REAL SQL Database to the Project Editor. Double-click on the this item in the Project Editor, and the Schema Editor for the database will appear. It is shown in Figure 2 on page 5.

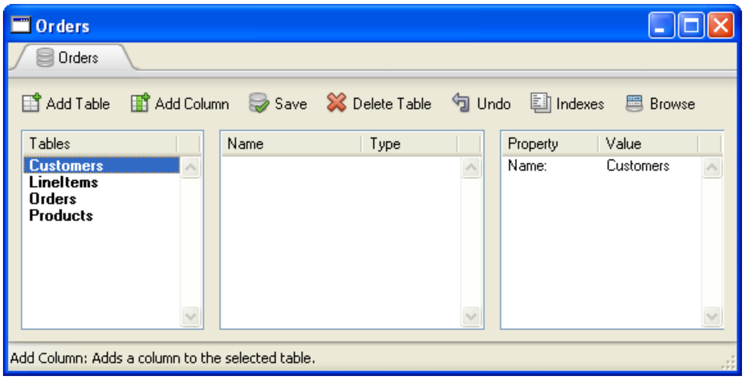
Figure 2. The Schema Editor for a new database.



Undo: Undo drop table

First, create the four tables used in the this database, Customers, Products, Orders, and LineItems. Do this by clicking the Add Table button and then setting the Name property of the new table to the table name. Figure 3 shows the Customers table being renamed in this way.

Figure 3. The four tables used in the Orders database example.

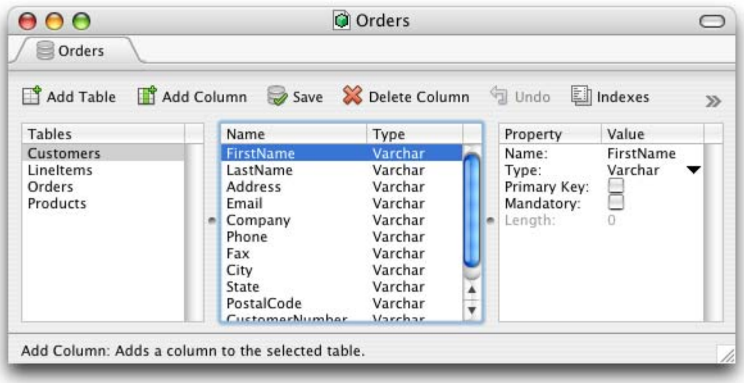


Property	Value
Name:	Customers

Add Column: Adds a column to the selected table.

To add a column to a table, select the table in the Table list on the left and then click the Add Column button. A new column is added to the column list in the middle pane of the editor. Change its name using the Name property and set the data type using the Type pop-up menu. The field schema (the information that defines the table) for the Customers table is shown in Figure 4 on page 6.

Figure 4. The field schema for the Customers table.



Tables
Customers
LineItems
Orders
Products

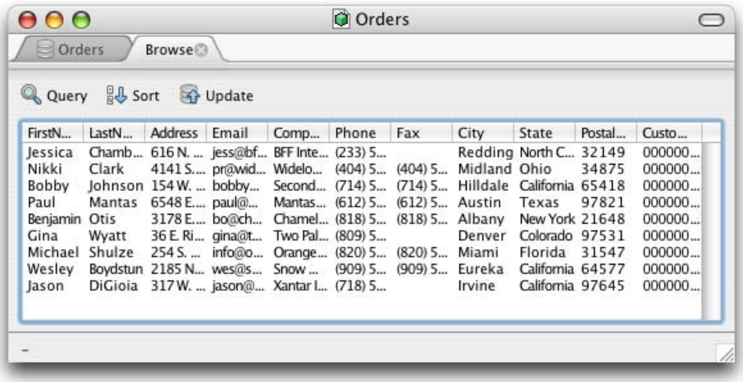
Name	Type
FirstName	Varchar
LastName	Varchar
Address	Varchar
Email	Varchar
Company	Varchar
Phone	Varchar
Fax	Varchar
City	Varchar
State	Varchar
PostalCode	Varchar
CustomerNumber	Varchar

Property	Value
Name:	FirstName
Type:	Varchar
Primary Key:	☐
Mandatory:	☐
Length:	0

Add Column: Adds a column to the selected table.

Built into the Database Schema Editor is a data viewer that can be used to view and modify data within a table by using Structured Query Language (SQL) statements, such as SELECT, UPDATE, and DELETE. If you are unfamiliar with SQL, see one of the many SQL reference books for a general background. Also check out http://www.sqlite.org for more information about the SQL used by REAL SQL Databases. The REAL SQL Database uses SQLite as its data source.

Figure 5. The data viewer for a table.



FirstN...	LastN...	Address	Email	Comp...	Phone	Fax	City	State	Postal...	Custo...
Jessica	Chamb...	616 N. ...	jess@bf...	BFF Inte...	(233) 5...		Redding	North C...	32149	000000...
Nikki	Clark	4141 S. ...	pr@wid...	Widelo...	(404) 5...	(404) 5...	Midland	Ohio	34875	000000...
Bobby	Johnson	154 W. ...	bobby...	Second...	(714) 5...	(714) 5...	Hilldale	California	65418	000000...
Paul	Mantas	6548 E. ...	paul@...	Mantas...	(612) 5...	(612) 5...	Austin	Texas	97821	000000...
Benjamin	Otis	3178 E. ...	bo@ch...	Chamel...	(818) 5...	(818) 5...	Albany	New York	21648	000000...
Gina	Wyatt	36 E. Ri...	gina@t...	Two Pal...	(809) 5...		Denver	Colorado	97531	000000...
Michael	Shulze	254 S. ...	info@o...	Orange...	(820) 5...	(820) 5...	Miami	Florida	31547	000000...
Wesley	Boydstun	2185 N...	wes@s...	Snow ...	(909) 5...	(909) 5...	Eureka	California	64577	000000...
Jason	DiGioia	317 W. ...	jason@...	Xantar I...	(718) 5...		Irvine	California	97645	000000...

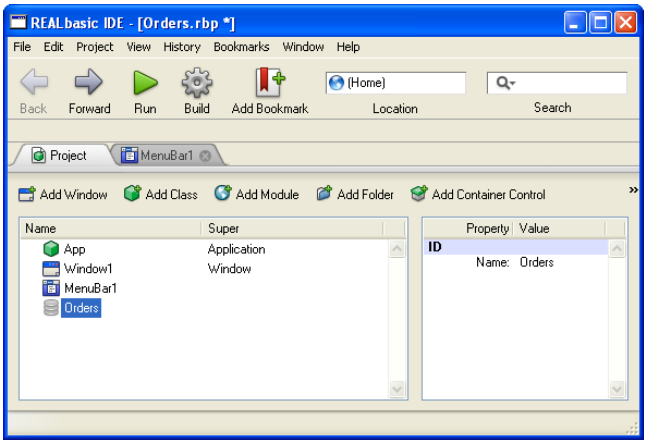
To open the browser for a table, select the table in the Tables list and then click the Browse button in the toolbar, shown in Figure 3 on page 5.

When a database is included as part of a project, the built application automatically connects to the database when it starts and provides global access to it through a global variable of type Database, with the variable name being the same as the name of the database.

One of the downsides to working with a database in this way (although handy for saving a little time and code) is that the application cannot switch to use another database while running, nor does it handle the situation well if the database is missing entirely.

Rather than create another database or warn the user, for example, the application will start up and run until at some point in the execution the code will cause an error when it is unable to connect to the database.

Figure 6. Embedding the Orders database in the project gives the programmer global access to the database through a variable of type Database named Orders.


REALbasic IDE - [Orders.rbp *]
File Edit Project View History Bookmarks Window Help
Back Forward Run Build Add Bookmark
(Home)
Location
Search
Project
MenuBar1
Add Window Add Class Add Module Add Folder Add Container Control

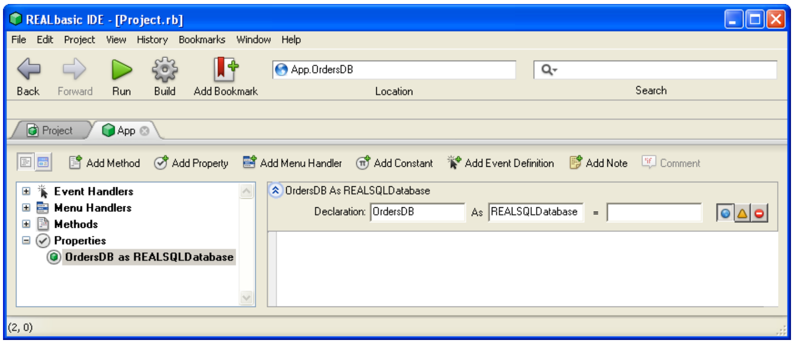
Name	Super
App	Application
Window1	Window
MenuBar1	
Orders	

Property	Value
ID	
Name:	Orders

Creating a Database In Code

The more commonly used method for creating and connecting to a database in REALbasic applications is to use the methods of the REALSQLDatabase class. This solution is a little more involved but is more flexible. In this example, a property of type REALSQLDatabase named OrdersDB is added to the App class that appears in the Project Editor.

Figure 7. The OrdersDB public property of the App class.


REALbasic IDE - [Project.rb]
File Edit Project View History Bookmarks Window Help
Back Forward Run Build Add Bookmark
App.OrdersDB
Location
Search
Project
App
Add Method Add Property Add Menu Handler Add Constant Add Event Definition Add Note Comment
Event Handlers
Menu Handlers
Methods
Properties
OrdersDB as REALSQLDatabase
(2, 0)

OrdersDB As REALSQLDatabase

Declaration: OrdersDB As REALSQLDatabase =

The code shown in Listing 1 on page 8 illustrates the major steps involved in creating a database and connecting to it from code. The first step is to create a new REALSQLDatabase class instance and set its DatabaseFile property to a FolderItem pointing to the location where either the database should already exist or otherwise be created.

If the database file doesn't exist (this is true when the application is run for the first time or the database was deleted for some reason), the application will attempt to create the database. The method executes the SQL statements that create the tables.

If the database file already exists, the program will attempt connect to it using the Connect method of the REALSQLDatabase class. If the Connect method returns False, an error occurred and the program will exit.

Listing 1. App class Open event.

```
// Our application is loading so the first thing that we want to do is
// connect to the database.  We have a property on the App class that
// will hold our database connection and can be accessed from anywhere
// in the application.  We will only connect to the database once and then
// we will continue to use that same connection throughout the project.

// Create database object
// OrdersDB is a property of the App class
OrdersDB = New REALSQLDatabase

// Set Database File
OrdersDB.databaseFile = GetFolderItem("Orders.rsd")

// Connect to the database
If OrdersDB.databaseFile.exists = True then
  // The database file already exists, so we want to connect to it.
  If OrdersDB.Connect() = False then
    // there was an error connecting to the database
    DisplayDatabaseError( False )
    Quit
    Return
  End if

Else
   // The database file does not exist so we want to create a new one.
   // The process of creating a database will establish a connection to it
   // so there isn't a need to call Database.Connect after we create it.
  CreateDatabaseFile
End if

// Set the application to AutoQuit,
// so if all windows are closed then the application will quit.
App.autoQuit = True
```

If the database file does not exist, the CreateDatabaseFile method is called. It contains the SQL CREATE TABLE commands to create the tables we need.

Listing 2. App.CreateDatabaseFile method.

```
// Create the database file
If OrdersDB.CreateDatabaseFile = False then
  // Error While Creating the Database
  MsgBox "Database Error" + EndOfLine + EndOfLine +_
          "There was an error when creating the database."
  Quit
End if

// Create the tables for the database
OrdersDB.SQLExecute "create table Customers (FirstName varchar,"+ _
   "LastName varchar, Address varchar, Email varchar, "+ _
  "Company varchar, Phone varchar, Fax varchar, City varchar, "+ _
   "State varchar, PostalCode varchar, "+ _
  "ID integer NOT NULL PRIMARY KEY)"

OrdersDB.SQLExecute "create table Orders (DateOrdered date,"+ _
     "DateShipped date, PurchaseOrder varchar, "+ _
     "ShipMethod varchar, Total double, SubTotal double, TotalTax
double,"+ _
    "TaxRate double, "OrderNumber integer NOT NULL PRIMARY KEY,"+ _
    "CustomerID integer)"

OrdersDB.SQLExecute "create table LineItems (ID integer NOT NULL "+ _
      "PRIMARY KEY, Price double, Quantity integer, "+ _
      "ProductName varchar, LineTotal double, PartNumber varchar,"+ _
      "OrderNumber integer)"

OrdersDB.SQLExecute "create table Products (Name varchar, "+ _
      "Price double, Image binary, PartNumber varchar)"
```

Finally, if there was an error connecting to the database, DisplayDatabaseError is called. It displays the error code and the error message. Optionally, it rolls back the database.

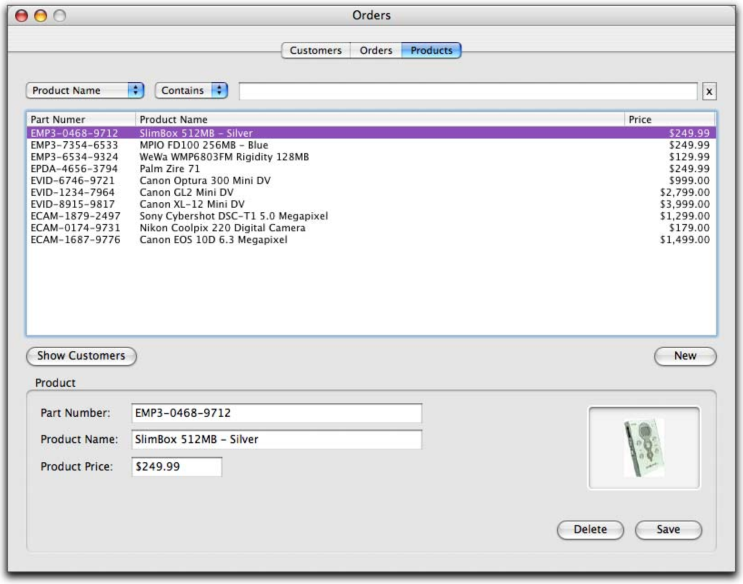
Listing 3. App.DisplayDatabaseError method.

```
Sub DisplayDatabaseError (doRollback as Boolean)
  // Display a dialog that shows information about the error that
  // has occurred from the database engine.  If requested a Rollback
  // will also happen on the database in order to undo any changes
  // that happened since the last commit.

  MsgBox "Database Error: " + str(OrdersDB.ErrorCode) + EndOfLine +_
        EndOfLine + OrdersDB.ErrorMessage

  // Rollback changes to the database if specified
  If doRollback then
    ordersDB.Rollback
  End if
```

Understanding the User Interface

The user interface for the Orders database is straightforward. It uses one window which has a large tab panel that contains three tabs, one for Customers, one for Orders, and the last one for Products.

Figure 8. The Products tab interface.



Part Numer	Product Name	Price
EMP3-0468-9712	SlimBox 512MB - Silver	$249.99
EMP3-7354-6533	MPIO FD100 256MB - Blue	$249.99
EMP3-6534-9324	WeWa WMP6803FM Rigidity 128MB	$129.99
EPDA-4656-3794	Palm Zire 71	$249.99
EVID-6746-9721	Canon Optura 300 Mini DV	$999.00
EVID-1234-7964	Canon GL2 Mini DV	$2,799.00
EVID-8915-9817	Canon XL-12 Mini DV	$3,999.00
ECAM-1879-2497	Sony Cybershot DSC-T1 5.0 Megapixel	$1,299.00
ECAM-0174-9731	Nikon Coolpix 220 Digital Camera	$179.00
ECAM-1687-9776	Canon EOS 10D 6.3 Megapixel	$1,499.00

Product

Part Number: EMP3-0468-9712

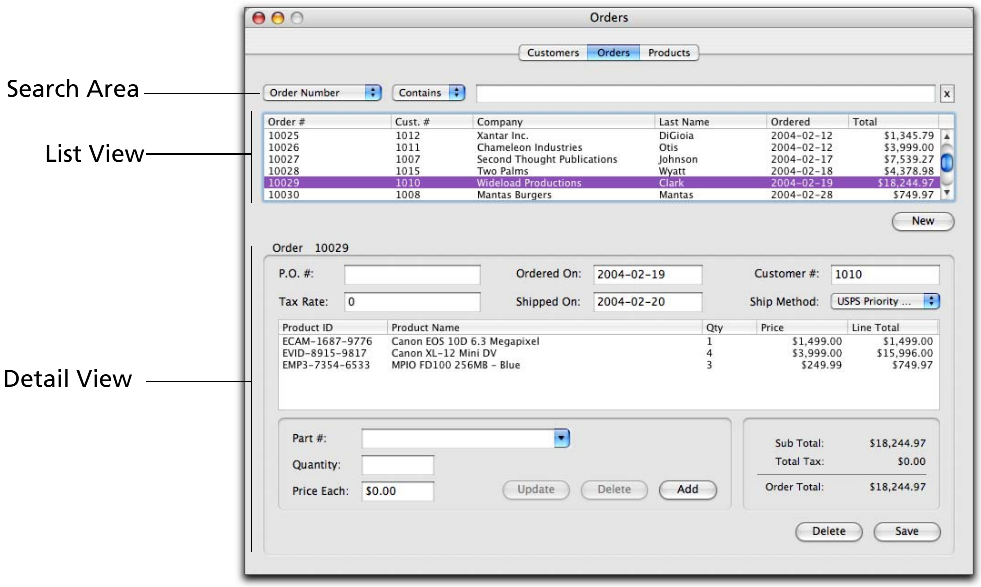
Product Name: SlimBox 512MB - Silver

Product Price: $249.99

Each tab contains what are referred to as a *List View* and a *Detail View* for the data related to each tab. The ListBox control at the top of each tab is the List View, and the GroupBox at the bottom of the window and all of the controls contained within it make up the Detail View. When a row is selected in the List View all of its information is shown within the Detail View.

Above the List View is a row of controls that is used to search the table. This makes finding and navigating to a specific record or records much easier than browsing through the list.

Figure 9. Identifying the List and Detail Views on the Orders tab.



Order #	Cust. #	Company	Last Name	Ordered	Total
10025	1012	Xantar Inc.	DiGioia	2004-02-12	$1,345.79
10026	1011	Chameleon Industries	Otis	2004-02-12	$3,999.00
10027	1007	Second Thought Publications	Johnson	2004-02-17	$7,539.27
10028	1015	Two Palms	Wyatt	2004-02-18	$4,378.98
10029	1010	Wideload Productions	Clark	2004-02-19	$18,244.97
10030	1008	Mantas Burgers	Mantas	2004-02-28	$749.97

Order 10029

P.O. #:
Ordered On: 2004-02-19
Customer #: 1010
Tax Rate: 0
Shipped On: 2004-02-20
Ship Method: USPS Priority ...

Product ID	Product Name	Qty	Price	Line Total
ECAM-1687-9776	Canon EOS 10D 6.3 Megapixel	1	$1,499.00	$1,499.00
EVID-8915-9817	Canon XL-12 Mini DV	4	$3,999.00	$15,996.00
EMP3-7354-6533	MPIO FD100 256MB – Blue	3	$249.99	$749.97

Part #:
Quantity:
Price Each: $0.00

Sub Total: $18,244.97
Total Tax: $0.00
Order Total: $18,244.97

The information for the record selected in the List View is automatically displayed in the Detail View.

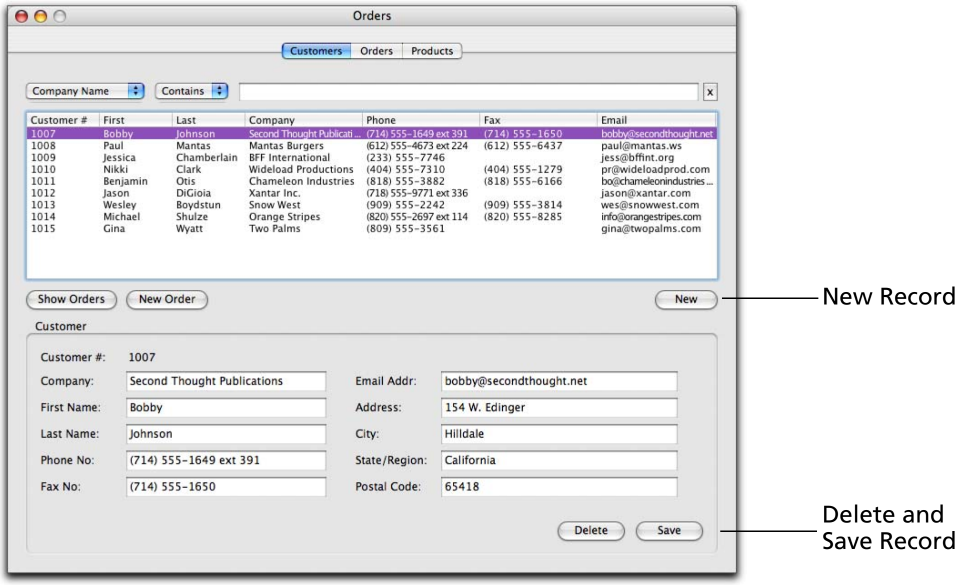
Clicking the New button just above the Detail View enables the controls within the Detail View groupbox. After entering the data for the new product, order, or customer, clicking the Save button will create a new database record and save it to the database.

When a record is selected in the List View, the controls in the Detail View are enabled, allowing changes to be made. When the user is finished making changes, he should then click the Save button to save the changes to the database back-end.

The Delete button removes the record selected in the List View from the database.

This dual-purpose nature of the Save button (assuming the roles of separate Add and Update buttons) is done with very simple programming and is an example of how to hide the inner workings of the program from the end-user for a better experience.

Figure 10. The New, Delete, and Save buttons on the Customers tab.



Customer #	First	Last	Company	Phone	Fax	Email
1007	Bobby	Johnson	Second Thought Publicati ...	(714) 555-1649 ext 391	(714) 555-1650	bobby@secondthought.net
1008	Paul	Mantas	Mantas Burgers	(612) 555-4673 ext 224	(612) 555-6437	paul@mantas.ws
1009	Jessica	Chamberlain	BFF International	(233) 555-7746		jess@bffint.org
1010	Nikki	Clark	Wideload Productions	(404) 555-7310	(404) 555-1279	pr@wideloadprod.com
1011	Benjamin	Otis	Chameleon Industries	(818) 555-3882	(818) 555-6166	bo@chameleonindustries ...
1012	Jason	DiGioia	Xantar Inc.	(718) 555-9771 ext 336		jason@xantar.com
1013	Wesley	Boydstun	Snow West	(909) 555-2242	(909) 555-3814	wes@snowwest.com
1014	Michael	Shulze	Orange Stripes	(820) 555-2697 ext 114	(820) 555-8285	info@orangestripes.com
1015	Gina	Wyatt	Two Palms	(809) 555-3561		gina@twopalms.com

Customer #:	1007		
Company:	Second Thought Publications	Email Addr:	bobby@secondthought.net
First Name:	Bobby	Address:	154 W. Edinger
Last Name:	Johnson	City:	Hilldale
Phone No:	(714) 555-1649 ext 391	State/Region:	California
Fax No:	(714) 555-1650	Postal Code:	65418

The other buttons above the Detail View, such as Show Orders or New Order on the Customers tab, trigger macros that perform a small number of actions, saving users a little bit of time and hassle. For example, clicking Show Orders when a customer is selected in the Customers tab List View switches to the Orders tab and enters the necessary information in the search area so that the results in the List View are only those orders that belong to the selected customer.

Opening the Main Window

When the application is launched, the App class's Open event runs. It either opens the existing database or creates a new database. Then the Open event of the MainWindow runs. It calls three methods which load the records from the Customers, Orders, and Products tables, populating the three List Views. The three methods that load the records are described in the sections "Adding Database Records" on page 28 and "Updating Database Records" on page 43.

If the application is running on Linux, it also tweaks the sizes of some of the controls so the interface looks right on Linux.

Listing 4. MainWindow.Open Event.

```
// The main window is being opened.
// Some controls on Linux must be taller than on other platforms
// in order to look correct.
#If TargetLinux then
  ResizeLinuxControls
#Endif

// We want to have all of the data lists display their initial
// data, which we will do by calling the UpdateList methods.

// Load the order records
UpdateOrderList

// Load the customer records
UpdateCustomerList

// Load the product records
  UpdateProductList
```

The ResizeLinuxControls method increases the height of all the RectControls in MainWindow. It does this by looping through all the controls in the window and casting them as RectControls. All RectControls have a Height property, so it can be changed with one generic line of code.

Listing 5. ResizeLinuxControls Method, part 1.

```
// On Linux some controls need to be taller than on other platforms in order to
// look right.  For example, a PushButton on Linux will have the captions cut off
// because it is too short at the current height.
// Instead of making all of the controls look extra big for non-Linux platforms we
// are going to have this function go through and resize all of the controls when
// we are running on Linux.  The function will look at all controls on the window
// and determine if it needs to be resized by looking at its type and current height.
// The window layout will already support the taller controls so the only property
// that will have to be adjusted is the control height.

// For Linux we are going to set the new control height to 28
Const kNewHeight = 28

Dim i as Integer
Dim ctl as RectControl
```

The main loop in ResizeLinuxControls examines all the controls in the window and casts them as RectControls. It is shown in Listing 6 on page 14.

Listing 6. ResizeLinuxControls Method, part 2.

```
// loop through all of the controls on the window
For i = 0 to controlCount - 1
// We can only resize RectControls so we can skip any control that is not one
  If NOT control( i ) isa RectControl then
    continue  // move to the next control in the loop
  End if

// The Window.Control array will return an object of the type Control.
// The Control class does not have the height property that we are interested
// in, but the RectControl class does.  In order to go from a Control to a
// RectControl we need to use typecasting.  After we have the object as a
// RectControl we can access the Height property.
    ctl = RectControl( control(i) )

  // The first criteria to determine if a control needs to be resized is to
  // check and see if the control height is below the new height.
  // Only controls that are smaller than the new height will have to be adjusted.
    If ctl.height < kNewHeight then

  // The next requirement is that the control is a type of control that we
  // want to resize. This includes PushButton, StaticText, EditField,
  // PopupMenu, and ComboBox.
    If (ctl isa PushButton) OR (ctl isa StaticText) OR (ctl isa EditField) OR _
      (ctl isa PopupMenu) OR (ctl isa ComboBox) then

      // The control meets all of the criteria for resizing, so now
      // we can set the height.
      ctl.height = kNewHeight

    End if

  End if
Next
```

Connecting the Database to the Interface

In this example, user does not need to create a database. When the application opens, the App class's Open event fires and connects to the database if it already exists. If it doesn't exist, a new database is created.

The user should also be able to delete the entire database file if they want and the program should simply create a new database and continue normally.

The App class is the instance of the Application class that is added to the Project Editor by default. As explained in the section "Creating a Database In Code" on page 7, the REALSQLDatabase object for the application (OrdersDB) is instantiated and the database file is created and connected to as needed.

The Open event of the App class is shown in Listing 1 on page 8. If the database does not exist, the CreateDatabaseFile method is called. It is shown in Listing 2 on page 9.

Displaying a List of Records

The Orders database uses a generic method to populate all of the ListBoxes in the user interface. It takes as parameters the ListBox to be populated and the RecordSet that contains the data to be displayed. It is called when the tab is displayed and whenever the ListBox needs to be repopulated, such as when the user does a search.

Listing 7. PopulateListBox method.

```
Private Sub PopulateListBox(lb as ListBox, rs as RecordSet)

// Populates the passed listbox with the data in the passed recordset
// This will loop through the records in the recordset and add rows
// to the listbox that contain the data in the recordset.

  Dim i as Integer
  // Clear the passed listbox
  lb.DeleteAllRows

  // Loop until we reach the end of the recordset
  While Not rs.EOF
    lb.AddRow ""// add a new row to the listbox

    // Loop through all of the fields in the recordset
    // and add the data to the correct column in the listbox
    For i = 1 to rs.FieldCount
      // The listbox Cell property is 0-based so we need to subtract 1
      //from the database field number to get the correct correct column
      //number.  This means field 1 is in column 0 of the listbox.

      lb.cell( lb.LastIndex, i-1 ) = rs.IdxField( i ).StringValue
    Next

    rs.MoveNext  // move to the next record
  Wend

// If the listbox is set to be sorted by a particular column then we want to
// sort the listbox contents after we populate it, so that they appear in the
// correct order.
  If lb.SortedColumn > -1 then  // the listbox is sorted by a column
   lb.sort  // sort the listbox data using the current sort settings
  End if
```

PopulateListBox is called by the UpdateCustomerList, UpdateLineItemsList, UpdateOrderList, and UpdateProductList methods. When MainWindow opens, its Open event fires and it calls UpdateOrderList, UpdateCustomerList, and UpdateProductList.

Managing PushButtons

Each screen uses pushbuttons that must be enabled or disabled depending on certain conditions. For example, the Products screen has a "Show Customers" button that displays the customers for the product that is selected in the Products ListBox. If there is no selection, then the Show Customers button should be disabled. Similarly, the Delete button should be enabled only when a product is selected.

When the window first opens and there is no selection, the buttons that require a selection should be disabled. This is done in the Open event handler for each Listbox. For example, the Open event handler for the ProductsList ListBox has the code:

```
If Me.ListIndex = -1 then // listbox has no selection
  ShowCustomers.Enabled=False
  DeleteProduct.Enabled=False
End if
```

Similar code is in the Open event handlers for the CustomersList, LineItemsList, and OrdersList ListBoxes.

The MainWindow uses a Private method, EnablePushbutton, to manage the enabled state of various buttons. It takes two parameters, a ListBox and the PushButton to be managed. Its code is:

```
Private Sub EnablePushbutton(lb as ListBox,pb as PushButton)
  // Enables the passed Pushbutton if the passed ListBox has a selection
  if lb.ListIndex = -1 then //no selection
    pb.Enabled=False
  Else
    pb.Enabled=True
  End if
```

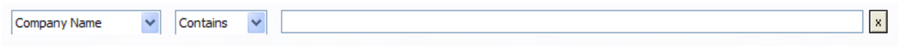
Each ListBox's Change event handler includes code that calls EnablePushButton for each PushButton that needs to be managed. For example, the Change event handler for the ProductsList ListBox includes the code:

```
// Enable Show Customers button
EnablePushButton(ProductsList,ShowCustomers)

//Enable Delete Product button
EnablePushButton(ProductsList,DeleteProduct)
```

Searching the Database

Searching is an important feature of every database application. Rather than complicate the interface with a separate search dialog and an additional window to display the results, the Orders database includes a search feature on each tab in the main window, just above the List View. On each tab in the project, four controls make up the search area.

Figure 11. The Search area above the List View in the Customers tab.



In the Customers tab, the first pop-up contains search fields such as the name of the company the customer works for, the customer's last name, and the products that customer has ordered. The pop-up just to the right of it contains the standard qualifiers such as "Contains" and "Begins With." The text field is where the information being searched for is entered, and the small BevelButton with an "X" is the Clear button. It clears the search and shows all the records.

Whenever any of the first three controls change (either of the pop-ups or the text field), they call the UpdateCustomerList method. It uses generic code to build the SQL SELECT statement that is run against the database.

This is done by constructing a valid SQL 'WHERE' clause from the search specifications that the user entered. The WHERE clause is appended it to the SELECT statement. Without the WHERE clause, the SELECT statement would return all records in the table.

The RecordSet that is returned is then passed to the PopulateListBox method (Listing 1 on page 8).

SQL does case-sensitive string comparisons, but the user may not enter the search string that matches the records' case correctly. Also, some records with the same content in the field may differ by case. If they were searching for a customer with the name "Jason DiGioia" and instead entered "Jason Digioia", they would unexpectedly find no results.

To allow for case-insensitive matching, the search string is transformed to consist of uppercase letters only using the built-in Uppercase() function in REALbasic, and then compared to all uppercase transformations of the records in the database using the SQL string function UPPER(). Comparing the two uppercase strings negates the case-sensitive string comparison, yielding the expected search results.

Listing 8. UpdateCustomerList method, part 1.

```
// Update the data in the customer list.
// We will build a query that contains any user-defined search filters
// and then display the records that match that criteria.

// This method is called when ever the contents of the Customer list needs
// to be updated or a new search term has been entered.

Dim sql, filter, searchTerm as String

// Take the string that the user has entered into the search field
// and prepare it to be used in a SQL command for searching.
// We want our searching to be case-insensitive, so the first step is to
// convert the string to have the same case. The queries will then be
// set up to compare all uppercase letters.
// After we have our search term we need to make sure that it can
// be used in a SQL command so we will pass it to the EscapeSQLData
// function.

// uppercase string for caseinsensitive searches
searchTerm = Uppercase( CustomerSearchFor.text )
// escape any special sql characters
searchTerm = EscapeSQLData( searchTerm )
```

The next section handles the case where the user is doing a search. In this case, the SQL query will have a WHERE clause. This section builds it.

Listing 9. UpdateCustomerList method, part 2.

```
// If the user has entered a search string then we will build a filter to be
// used in the SQL statement.
If searchTerm <> "" then
  // Search by Customer's Orders
  If CustomerSearchBy.Text = "Products Ordered" then
    // Build a subselect that will return the matching customer IDs
    // by looking at LineItems
    filter = "ROWID IN (SELECT DISTINCT C.ROWID FROM Customers C,"+ _
      "Orders O, LineItems L WHERE"+ _
      " O.CustomerID = C.ID AND L.OrderNumber = O.OrderNumber"+ _
      " AND upper(L.ProductName) LIKE '%" + searchTerm + "%')"
  else
    // First we need to determine which database column
    // the search is going to be performed on.
    Select Case CustomerSearchBy.Text
    Case "Company Name"
      filter = "Company"
    Case "Last Name"
      filter = "LastName"
    Case "City"
      filter = "City"
    Case "State/Region"
      filter = "State"
    Case "Postal Code"
      filter = "PostalCode"
  End Select
```

The next section of code builds the filter by converting the expression to uppercase and adding the SQL 'LIKE' keyword and the '%' wildcard.

Listing 10. UpdateCustomerList method, part 3.

```
// Since we converted the search term to be all uppercase
// in order to perform a case insensitive search, we also need to tell the
// database engine to compare against the uppercased data in
// the column.  To do this we use the SQL function 'upper'.
// The syntax of the SQL command will
// look like this:  select * from table where upper(column) = 'TEST'
// Enclose the column name in the upper() function

  filter = "upper("+ filter +")"

// Add the actual search term and wildcard characters to the filter
// according to the user settings.
  Select Case CustomerSearchOption.Text
    Case "Contains"
      filter = filter + " LIKE '%" + searchTerm + "%'"
    Case "Is Exactly"
      filter = filter + " = '" + searchTerm + "'"
    Case "Begins With"
      filter = filter + " LIKE '" + searchTerm + "%'"
    Case "Ends With"
      filter = filter + " LIKE '%" + searchTerm + "'"
  End Select

  End if
End if
```

The last section of this method creates the SQL statement, runs it against the database, and then passes the returned recordset to PopulateListbox to display the records.

Listing 11. UpdateCustomerList method, part 4.

```
// Build the SQL statement that will be used to select the records
sql = "SELECT ID, FirstName, LastName, Company, Phone, Fax, Email"+ _
        "FROM Customers"

  // Add the filter to the SQL statement if we have one.
If filter <> " " then
  sql = sql +" WHERE "+ filter
end if

// Now we select the records from the database and add them to the list.

Dim rs as RecordSet
rs = App.ordersDB.SQLSelect( sql )
PopulateListBox customersList, rs
rs.Close
```

The EscapeSQLData function that is called whenever this method is building a SQL query handles any embedded quotes that the user may have entered into the search string. The single quote is interpreted as the string delimiter by SQL, so it must be escaped.

Listing 12. EscapeSQLData function.

```
EscapeSQLData(data as String) as Sring
// Prepare a string for use in a SQL statement.  A string which is being
// placed into a SQL statement cannot have a single quote in it since that will
// make the database engine believe the string is finished.
// For example the word "can't" will not work in SQL because
// it will see the word as just "can".
// In order to get around this you must escape all single quotes
/ /by adding a second one.  So "can't" will become "can''t"
// and then the SQL command will work.

// Replace all single quotes with two single quote characters

data = ReplaceAll( data, "'", "''" )

  // Return the new data which is ready to be used in SQL
Return data
```

The first step in building the query is to check whether the user entered a search specification. If not, then the default query is run, requesting all records. If there is text in the field, and the first pop-up menu's selection is the Products Ordered item, the query built will select all of the customers where any of the products they ordered contains the text in the filter field, allowing the user to find the customers who ordered the entered product.

If the selected field is not Products Ordered, the SQL statement is instead built to compare the search field's text to the selected field. For example, if Company Name is selected in the pop-up, the SQL statement will search the Company field of the Customers table for the entered string.

With the SQL "LIKE" qualifier and the "%'" wildcard, the SQL statement is made to logically match the Contains, Is Exactly, Begins With, or Ends With selection in the CustomerSearchOption pop-up.

After determining which database column to search and how to compare the search string to the column in the table's records, the SQL statement is constructed and passed to the SQL SELECT statement. The RecordSet that is returned is then passed to PopulateListBox.

This same format and logic is used to construct customized queries for the Orders and Products tabs in the UpdateOrderList and UpdateProductList methods.

Displaying Detailed Record Information

The Detail View is populated with data whenever a row from the List View is selected. Because of this relationship, the usual location for loading the data into the Detail View is in the Change event of List View's ListBox or in a method called from it.

In the Change event of each List View's ListBox (CustomersList, OrdersList, and ProductsList), either of the following will happen: If no row is selected, the Detail View's GroupBox is disabled and all of the fields and controls in the Detail View are reset or cleared. If a row is selected in the List View, the GroupBox is enabled and all of the Detail View's controls are populated with the data.

Listing 13. OrdersList Change, part 1.

```
// Display information about the selected order record
Dim rs as RecordSet
Dim i as Integer

// Clear the order controls
ClearOrderFields

// Enable order edit controls if an order record is selected
OrderGroupBox.Enabled = (Me.ListIndex <> -1)

// Enable Delete Order button
EnablePushButton(OrdersList,DeleteOrder)

// Update the line items list
UpdateLineItemsList

// If a record is not selected then we do not have anything else to do
If Me.listIndex = -1 then
  Return
End if

// Get all of the information from the selected order record
rs = App.OrdersDB.SQLSelect("SELECT * FROM Orders WHERE "+ _
          "OrderNumber = '" + Me.Cell(Me.listIndex,0) + "'")
If App.ordersDB.error then
  App.DisplayDatabaseError False
  Return
End if
```

The next section of code loads the data into the Detail View.

Listing 14. OrdersList Change, part 2.

```
// Display the order record data
StatOrderNumber.Text = rs.Field("OrderNumber").StringValue
OrderOrderedOn.Text = rs.Field("DateOrdered").StringValue
OrderShippedOn.Text = rs.Field("DateShipped").StringValue
OrderPurchaseOrder.Text = rs.Field("PurchaseOrder").StringValue
OrderTaxRate.Text = Str(rs.Field("TaxRate").DoubleValue)
StatSubTotal.Text = Format(rs.Field("SubTotal").DoubleValue,_
          "\$###,###,##0.00")
  StatTotalTax.Text = Format(rs.Field("TotalTax").DoubleValue,_
          "\$###,###,##0.00")
  StatOrderTotal.Text = Format(rs.Field("Total").DoubleValue,_
          "\$###,###,##0.00")
  OrderCustomerNumber.Text = rs.Field("CustomerID").StringValue

// Loop through the items in the ShipMethod popup menu
// and find the value which matches that of the order.
OrderShipMethod.listIndex = -1// reset the selection
For i = 0 to OrderShipMethod.listCount-1
  If OrderShipMethod.list(i) = rs.Field("ShipMethod").StringValue then
  // found matching item
    OrderShipMethod.listIndex = i  // set selected item
    Exit
    // we found the match so we don't need to look at the remaining items
  End if
Next

// Close the recordset
rs.Close
```

Since the first column in the Orders tab List View is the order number for the order and each order number is unique, the order number is used to do the lookup.

All of the fields in the database are selected using the SELECT statement so that they can be displayed in the Detail View. Controls in the Detail View are then populated with the selected record in the ListBox.

Filling in the Detail View for the Orders tab is a little more involved than the Customers and Products tabs since the line items for the selected order must also be retrieved from the Line Items table and displayed in their own listbox.

Using a separate RecordSet instance (named "lineSet" in the SaveOrder Action event handler), the line items associated with the order are selected from the LineItems table by selecting line item records that have the selected row's order number in the Order-Number field,. They are displayed within the LineItemsList ListBox in the Detail View.

Notice that the CellTag property of the first cell in each row in the LineItemsList ListBox is used to store the ID of the line item. Because the interface of the application allows the user to modify the line items, a “hook” is needed in order to know which line item record relates to which line item in the ListBox so that they can be updated properly.

The format of the code to retrieve and display data in the Detail View for the Products and Customers tabs is very much the same. It is simpler as they do not have an embedded List View that display data from another table.

Checking for Required Fields

One of the essential aspects of data entry is making sure that the data being entered is complete and valid. In the case of the Orders example, it is necessary make sure that all of the fields that are required have values before adding a new record or modifying an existing record.

Rather than writing separate specialized code for each table and have multiple copies of identical code for displaying an error message if a required field is empty, the database uses a small reusable chunk of code that calls a generic method to determine whether the required fields have data in them. The method takes an array of RectControls to be checked and a String array of field labels.

The method creates a MessageDialog that displays an error message explaining which required fields are empty or missing. The values of the Fields array are the required fields that the method will examine, and the Labels array are the names of the fields, such as Product Name.

Listing 15. CheckMandatoryFields Function, part 1.

```
CheckMandatoryFields(Fields() as RectControl,Labels() as String as Boolean
// This function checks to make sure all required data has been provided
// and will display an error dialog if some data is missing.
// This function takes an array of controls which are required to have data
// and an array of field labels to describe the type of data
// that is supposed to be in these controls.  The labels
// are used to display errors to the user when required data is not found.

  Dim i as Integer
  Dim dlog as MessageDialog
  Dim isEmpty, errorFound as Boolean

 // Create the error dialog
  dlog = New MessageDialog
  dlog.ActionButton.Caption = "OK"
  dlog.CancelButton.Visible = False
  dlog.Message = "Missing Required Fields"
  dlog.Explanation = "There are empty required fields that must be "+ _
      "filled in before adding "the record to the database. These fields"+ _
      " include: "
```

The next section of code loops through the array of RectControls. It needs to cast each RectControl so that it can do the proper test for missing values.

Listing 16. CheckMandatoryFields Function, part 2.

```
// Loop through each Control
  For i = UBound(Fields) downTo 0

// Depending on what kind of control it is, do a test to determine whether
// it's empty or not.
// Since we have an array of RectControls we will have to TypeCast each
// RectControl into the appropriate class in order to perform the data check.
  if Fields(i) IsA EditField then// check editfields
    isEmpty = (EditField( Fields(i) ).Text = " ")

  elseif Fields(i) IsA ComboBox then// check comboboxes
    isEmpty = (ComboBox( Fields(i) ).Text = " ")

  elseif Fields(i) IsA ListBox then// check listboxes
      isEmpty = (ListBox (Fields(i) ).ListCount = 0)
  End if

    If isEmpty then
    // Data check has failed so we should add this control to the list
    // of missing data.
      dlog.Explanation = dlog.Explanation + EndOfLine + "     - " + Labels(i)

      // Will Need to show the Dialog
      errorFound = True
    End if
  Next

 // Show the Dialog If Needed
  If errorFound then
    Call dlog.ShowModal // don't need the result

  // Everything is Not OK
  Return False
End if

// Everything is OK
Return True
```

To determine whether a field is empty or not, the method loops through each of the indexes in the array, determines what kind of control it is (whether it is an EditField, a ComboBox, or a ListBox) and uses a simple test to determine if it is empty. For example, in the case of an EditField or a ComboBox, the field is empty if its Text property is empty.

When a field is determined to be empty, the boolean ErrorFound variable is set to True, signaling the following If statement to display the Message Dialog with the relevant error messages.

After setting up the dialog, it is displayed only if at least one field was empty. When the dialog is closed, the method will return False signaling to the calling method that not every field was given a value and that the operation (updating or creating the record) should be aborted. If all mandatory fields were filled in, the CheckMandatoryFields() method will return True.

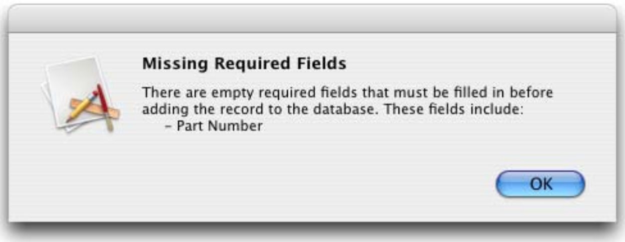
Listing 17 shows an example of how the CheckMandatoryFields function is called. In the SaveProduct button's Action Event, the required fields are the Part Number, Product Name, and Product Price. The Fields parameter to the CheckMandatoryFields method is an array containing those fields and the Labels array contains "Part Number", "Product Name", and "Product Price".

Listing 17. Calling code from the SaveProduct.Action Event.

```
Dim Fields(-1) as RectControl, Labels(-1) as String
Fields.append ProductPartNumber
Fields.append ProductName
Fields.append ProductPrice
Labels = Array("Part Number", "Product Name", "Product Price")
If Not CheckMandatoryFields(Fields, Labels) then
  Return  // all required data has not been provided
End if
```

If a user attempts to add or update a product but forgets to define a part number for the product, an error message such shown in Figure 12 is displayed.

Figure 12. An error dialog is displayed when one or more mandatory fields are empty.


Missing Required Fields

There are empty required fields that must be filled in before adding the record to the database. These fields include:
 - Part Number

Adding Database Records

A database application isn't of much use to anyone unless there is a way to add records to it. In REALbasic, there are two ways of adding a new record to a database. The first is to insert a DatabaseRecord instance into the table using the InsertRecord method of the Database class. The second is to use a SQL statement such as:

```
INSERT INTO Products (PartNumber, ProductName, Price) VALUES
('EPDA-4656-3794', 'Palm Zire 71', 249.99)
```

that is passed to the database engine via a SQL SELECT statement.

While both are acceptable and are used by REALbasic developers in their projects, using a DatabaseRecord is probably the more common solution because it's a little easier to not have to work with SQL statements.

To add a new record to a table using the first technique, create a DatabaseRecord instance and fill it with the field values for the record. Then use the InsertRecord method to add the record to a table.

Adding a Customer Record

The code from the Customers tab's Save button is shown in Listing 18 on page 30. Because the Save buttons on each of the tabs perform the dual purpose of adding and updating a record, the first step is to determine which of the two actions needs to be done. The Orders database does this by looking a the Customer Number field.

Since a customer's Customer Number is assigned from code when the record is created (as opposed to the user's creating a customer number for each customer), if a customer is selected in the customer's list, that customer's ID would be displayed in the StatCustomerNumber StaticText field.

Listing 18. SaveCustomer.Action, part 1.

```
// Saves changes to the customer record.  If an existing customer record was
// modified then we will update that record, otherwise we will insert a new
// record.  The CustomerID will be used to determine if an existing record
// was modified or if this is a new record.

Dim rec as DatabaseRecord
Dim rs as RecordSet

// First we are going to verify that all of the required fields have data
// in them.  We do this by adding the required controls to an array and then
// passing that to the CheckMandatoryFields function.
dim Fields(-1) as RectControl
dim Labels(-1) as String
Fields.append CustomerFirst
Fields.append CustomerLast
Fields.append CustomerPhone
Fields.append CustomerAddress
Fields.append CustomerCity
Fields.append CustomerState
Fields.append CustomerPostalCode
Labels = Array("First Name", "Last Name", "Phone Number", "Address",_
         "City", "State/Region", "Postal Code")
If not CheckMandatoryFields(Fields, Labels) then
  Return// all required data has not been provided
End if
```

If the StatCustomerNumber field is empty when the Save button is clicked, it is a new record. The data in the fields within the Detail View should be inserted into a new record and then added to the Customers table. REAL SQL Database automatically generates a new ID for the record that is used as the primary key field. If the StaticText field is not empty, it is an existing record and it should be updated.

Listing 19. SaveCustomer.Action, part 2.

```
 // Update customer data
If StatCustomerNumber.Text = "" then
  // If we don't have a Customer ID then we need to insert a new record.

  // Create a new database record with all of the customer information
  rec = New DatabaseRecord
  rec.Column("Company") = CustomerCompany.Text
  rec.Column("FirstName") = CustomerFirst.Text
  rec.Column("LastName") = CustomerLast.Text
  rec.Column("Phone") = CustomerPhone.Text
  rec.Column("Fax") = CustomerFax.Text
  rec.Column("Email") = CustomerEmail.Text
  rec.Column("Address") = CustomerAddress.Text
  rec.Column("City") = CustomerCity.Text
  rec.Column("State") = CustomerState.Text
  rec.Column("PostalCode") = CustomerPostalCode.Text

  // Insert Record Into table "Customers" of the Database
  App.OrdersDB.InsertRecord "Customers", rec

  // Display Error if one occurred
  If App.OrdersDB.Error then
    App.DisplayDatabaseError True
    Return
  End if
```

In the code in Listing 19, a DatabaseRecord is created and the field values are assigned. The parameter to the Column method of the DatabaseRecord instance is the name of the column to which the value after the equal sign is assigned.

There are other versions of the Column function for different data types. For example, there is an IntegerColumn method and a DoubleColumn method for storing values in integer and floating point columns. See the DatabaseRecord class in the *Language Reference* for more information.

The code in Listing 20 on page 32 handles the case in which the StatCustomerNumber field contains data. In this case, the Customers tab is displaying an existing record and the user wants to save updates.

Listing 20. SaveCustomer.Action, part 3.

```
else  // We have a customer id so we should update the existing record

  // Find the existing record by searching for the customer id
  rs = App.OrdersDB.SQLSelect("SELECT * FROM Customers WHERE ID = '" +_
        EscapeSQLData(StatCustomerNumber.Text) + "'")
  If App.ordersDB.error then
    App.displayDatabaseError True
    Return
  End if
  // Update the existing database record.  First we need to make the
  // RecordSet editable which will try to obtain a lock on the record.
  // Be sure to check for errors after calling RecordSet.Edit in case
  // something went wrong here.
  rs.Edit
  If App.ordersDB.error then
    App.displayDatabaseError True
  End if

  // Update the customer information
  rs.Field("Company").StringValue = CustomerCompany.Text
  rs.Field("FirstName").StringValue = CustomerFirst.Text
  rs.Field("LastName").StringValue = CustomerLast.Text
  rs.Field("Phone").StringValue = CustomerPhone.Text
  rs.Field("Fax").StringValue = CustomerFax.Text
  rs.Field("Email").StringValue = CustomerEmail.Text
  rs.Field("Address").StringValue = CustomerAddress.Text
  rs.Field("City").StringValue = CustomerCity.Text
  rs.Field("State").StringValue = CustomerState.Text
  rs.Field("PostalCode").StringValue = CustomerPostalCode.Text

  // Update the record in the database
  rs.Update
  If App.OrdersDB.Error then// handle errors
    App.DisplayDatabaseError True
    Return
  End if
End if

// Commit changes to the database
App.OrdersDB.Commit

// Update the list of customers
UpdateCustomerList
```

After checking for any errors and saving the changes, the List View of the Customers tab is refreshed by calling UpdateCustomerList.

Adding an Order Record

Creating a new order record and inserting it into the Orders table is very similar to doing so with a customer record. The major difference is that line items for the order are stored in their own related table.

The same technique to determine whether a new record should be created or an existing record should be updated is used here. The StatOrderNumber StaticText field will contain an order number only if an existing order has been selected in the List View.

As with the other update methods, SaveOrder does the same checking for mandatory fields. Then it uses a function to convert the totals to Double values.

Listing 21. SaveOrder.Action, part 1.

```
Dim rec as DatabaseRecord
Dim i, count, orderNumber as Integer
Dim lineItemID as String
Dim subTotal, taxTotal, total as Double
Dim rs, lineSet as RecordSet

Check the mandatory fields, as was done for SaveCustomer
Dim Fields(-1) as RectControl, Labels(-1) as String
Fields.append OrderCustomerNumber
Fields.append OrderTaxRate
Fields.append LineItemsList
Labels = Array("Bill To", "Tax Rate", "Line Items")
If Not CheckMandatoryFields(Fields, Labels) then
  Return// all required data has not been provided
End if

// Get totals by stripping out characters from the strings in the EditFields
// Store order totals in numeric variables
subTotal = moneyToDouble( statSubTotal.text )
taxTotal = moneyToDouble( statTotalTax.text )
total = moneyToDouble( statOrderTotal.text )
```

Since the data for an order is stored in two tables, the code is enclosed in a SQL transaction to guard against an incomplete data entry. You would not want line items saved without the parent order record or vice versa.

Listing 22. SaveOrder.Action, part 2.

```
// Start a new transaction
App.ordersDB.sqlExecute "begin transaction"

// Modify the order in the database
If StatOrderNumber.Text = "" then
// There isn't an order number so this means that we need
// to insert a new record into the database for this order.

// Create a database record for the new order
  rec = New DatabaseRecord
  rec.Column("PurchaseOrder") = OrderPurchaseOrder.Text
  rec.Column("CustomerID") = OrderCustomerNumber.Text
  rec.Column("DateOrdered") = OrderOrderedOn.Text
  rec.Column("DateShipped") = OrderShippedOn.Text
  rec.Column("ShipMethod") = OrderShipMethod.Text
  rec.DoubleColumn("TaxRate") = Val(OrderTaxRate.Text)
  rec.DoubleColumn("SubTotal") = subTotal
  rec.DoubleColumn("TotalTax") = taxTotal
  rec.DoubleColumn("Total") = total

  // Insert the new record into the database
  App.OrdersDB.InsertRecord "Orders", rec
  if App.OrdersDB.Error then// handle errors
    App.DisplayDatabaseError true
    Return
  End if

// Get the order number for the order that was just entered
// into the database.
// The OrderNumber column is the primary key for the Orders table and
// the REALSQLDatabase.LastRowID function will return the ID number
// for the last row that was inserted into the database.
// In this case it will be for the order record that we just inserted above.
  orderNumber = App.ordersDB.lastRowID()
```

The ‘else’ clause of the If statement handles the case where we are working with an existing record. This section of code updates the existing record.

Listing 23. SaveOrder.Action, part 3.

```
else
// We have an order number so we should update the existing order record.

// Get the order number of the selected Order
  orderNumber = Val( StatOrderNumber.Text )

  // Select the existing order record that we are updating
  rs = App.OrdersDB.SQLSelect("SELECT * FROM Orders WHERE " +_
       "OrderNumber = '" + str(orderNumber) + "'")

  // Update the existing database record.  First we need to make the
  // RecordSet editable which will try to obtain a lock on the record.
  // Be sure to check for errors after calling RecordSet.Edit in case
  // something went wrong here.
  rs.Edit
  If App.ordersDB.error then
    App.displayDatabaseError False
    Return
  End if

// Update the database record
  rs.Field("OrderNumber").StringValue = StatOrderNumber.Text
  rs.Field("PurchaseOrder").StringValue = OrderPurchaseOrder.Text
  rs.Field("CustomerID").StringValue = OrderCustomerNumber.Text
  rs.Field("DateOrdered").StringValue = OrderOrderedOn.Text
  rs.Field("DateShipped").StringValue = OrderShippedOn.Text
  rs.Field("ShipMethod").StringValue = OrderShipMethod.Text
  rs.Field("TaxRate").DoubleValue = Val(OrderTaxRate.Text)
  rs.Field("SubTotal").DoubleValue = subTotal
  rs.Field("TotalTax").DoubleValue = taxTotal
  rs.Field("Total").DoubleValue = total

// Update the order record in the database
  rs.Update
  If App.OrdersDB.Error then
    App.DisplayDatabaseError True
    Return
  End if

End if
```

The next section of code updates the line items in the order.

Listing 24. SaveOrder.Action, part 4.

```
// The order record has been updated so now we need to make all changes
// to the line items on this order.  We will loop through all of the line items
// an update the data or insert a new record where an existing one does not
// already exist.

// Loop through all of the line items
For i = 0 to LineItemsList.ListCount-1

// Look for a LineItemID in the CellTag of the first column.
// If one is found then a line item record already exists in the database,
// but if we don't find one then we need to create a new line item record.

  LineItemID = LineItemsList.CellTag( i, 0 )

// Update the existing line item when we have an ID
  If LineItemID <> "" then
    lineSet = App.OrdersDB.SQLSelect("SELECT * FROM LineItems " +_
        "WHERE ID = '" + lineItemID + "'")
    lineSet.Edit// make the recordset editable
    If App.ordersDB.error then
      App.displayDatabaseError true
      Return
    End if

    // update the data in the line items record
    lineSet.Field("OrderNumber").StringValue = StatOrderNumber.Text
    lineSet.Field("PartNumber").StringValue = LineItemsList.Cell(i, 0)
    lineSet.Field("ProductName").StringValue = LineItemsList.Cell(i, 1)
    lineSet.Field("Quantity").StringValue = LineItemsList.Cell(i, 2)
    lineSet.Field("Price").StringValue = LineItemsList.Cell(i, 3)
    lineSet.Field("LineTotal").StringValue = LineItemsList.Cell(i, 4)

  // update the line item record in the database
    lineSet.Update
    If App.ordersDB.error then
      App.displayDatabaseError true
      Return
    End if
```

The next section of code handles the case where the LineItemID has not been assigned and the line item needs to be added.

Listing 25. SaveOrder Action, part 5.

```
  else
  // Create a new line item record
    rec = New DatabaseRecord
    rec.Column("OrderNumber") = StatOrderNumber.Text
    rec.Column("PartNumber") = LineItemsList.Cell(i, 0)
    rec.Column("ProductName") = LineItemsList.Cell(i, 1)
    rec.IntegerColumn("Quantity") = Val(LineItemsList.Cell(i, 2))
    rec.DoubleColumn("Price") = Val(LineItemsList.Cell(i, 3))
    rec.DoubleColumn("LineTotal") = Val(LineItemsList.Cell(i, 4))

    // Insert new record into the database
    App.OrdersDB.InsertRecord "LineItems", rec
    If App.ordersDB.error then
      App.displayDatabaseError True
      return
    End if

  End if
Next
```

The next section of code handles any line items that were deleted in the course of the transaction. The IDs of those line items are temporarily stored in the RemovedLineItems array.

Listing 26. SaveOrder.Action, part 6.

```
// Now we need to delete any line item records that were marked
// to be removed.
// When a line item is deleted the record id is stored in an array, which we
// will loop through and delete all line items that we find in that array.

For Each lineItemID in RemovedLineItems
  App.OrdersDB.SQLExecute("DELETE FROM LineItems WHERE " +_
        "ID = '" + lineItemID + "'")
  If App.OrdersDB.Error then// handle errors
    App.DisplayDatabaseError True
    Return
  End if
Next
```

Finally, all the changes are committed to the database via a call to Commit and the list is updated with a call to UpdateOrderList.

Listing 27. SaveOrder.Action, part 7.

```
// Commit changes to the database
App.OrdersDB.Commit

// Update the list of orders
UpdateOrderList
```

Creating line item records for the order is done in the same way as for other tables. While looping through the LineItemsList ListBox, a new DatabaseRecord is created and inserted into the LineItems table. Notice that the order number for the newly created order is assigned to the OrderNumber column of the record.

Adding a Product Record

When you add a new product record, the database must check the Read-only property of the ProductPartNumber EditField to determine whether the record is new.

When a new product is being created, the product's part number is assigned by the user rather than by the database. Once a part number has been assigned, it cannot be changed, as it will break the link to existing orders that refer to the product.

To handle this, if the New product button was clicked, the ProductPartNumber field's ReadOnly property is set to False so that the user can enter a value. However, if a product is selected in the List View, the ReadOnly property is set to True so that the part number displayed in the form cannot be edited.

The SaveProduct Action event handler begins by checking that all required fields are filled in. This is done by calling CheckMandatoryFields with the list of the required fields.

Listing 28. SaveProduct.Action, part 1.

```
Dim imageData as String
Dim rec as DatabaseRecord
Dim rs as RecordSet
Dim bs as BinaryStream
Dim f as FolderItem

// First we are going to verify that all of the required fields have data
// in them.  We do this by adding the required controls to an array
// and then passing that to the CheckMandatoryFields function.  See the
// CheckMandatoryFields function for more information.

Dim Fields(-1) as RectControl, Labels(-1) as String
Fields.append ProductPartNumber
Fields.append ProductName
Fields.append ProductPrice
Labels = Array("Part Number", "Product Name", "Product Price")
If not CheckMandatoryFields(Fields, Labels) then
  Return  // all required data has not been provided
End if
```

Another difference between the Products table and the others is that it stores a picture. Each product record has an image associated with it. Because there is no Picture data type in REAL SQL Database, it is necessary to convert the image into a data type that the database handles. The REAL SQL Database uses Blob fields to store images.

To save an image, it is first saved to a temporary file using the FolderItem class's SaveAsJPEG method (You could also use SaveAsPicture if you wanted to use the platform's native picture file format.). Then it is read into a String with a BinaryStream. The string is saved in a Blob column in the Products table.

The next section of code saves the image by saving it to a temporary file and then reading it in as binary data.

Listing 29. SaveProduct.Action, part 2.

```
// Convert the picture to binary data that can be stored in the database.
// We do this by saving the picture to a temporary file and then
// reading it back in as binary data.

If productImageWell.image <> NIL then
  // Get a temporary file to save the image to
  f = TemporaryFolder.Child( "Temp_Image.jpg" )

  // Save the image out to the file
  f.saveAsJPEG productImageWell.image

  // Open the file as a BinaryStream and read the data in
  bs = f.openAsBinaryFile( False )
  If bs <> NIL then
    imageData = bs.read( bs.length )
    bs.close
  End if

  // delete the temporary file if it exists
  If f.exists then
    f.delete
  End if
End if
```

Also, when adding a new record, the user-entered part number must be checked against existing products to ensure that it is unique, as this is also a requirement. To check for a duplicate part number the SaveProduct.Action event handler gets the number of existing products with that part number.

Listing 30. SaveProduct.Action, part 3.

```
// The only time that the Part Number field is editable is when we are
// entering a new product.  When an existing product is being modified
// then the part number field is read-only so that the part number cannot
// be changed.

If ProductPartNumber.ReadOnly = False then

// Since we are allowed to edit the part number then we must be
// entering a new record.
// Check to see if the Part Number is unique by counting the number
// of products that already have this part number.
  rs = App.OrdersDB.SQLSelect("SELECT count(*) FROM Products WHERE" +_
        "PartNumber = '" + EscapeSQLData(ProductPartNumber.Text) + "'")
  If rs.idxField( 1 ).IntegerValue > 0 then
    // record exists if the count is greater than 0
    MsgBox "Duplicate Product" + EndOfLine + EndOfLine + _
            "A product with this part number already exists."
    Return
  End if
```

The next section of code creates a new record for the product. It stores the image into a Blob column by storing the string that was read with the BinaryStream object.

Listing 31. SaveProduct.Action, part 4.

```
// Create a new database record for the product
  rec = New DatabaseRecord
  rec.Column("PartNumber") = ProductPartNumber.Text
  rec.Column("Name") = ProductName.Text
  rec.DoubleColumn("Price") = moneyToDouble( productPrice.text )
  rec.BlobColumn("Image") = imageData

  // Insert product record into the database
  App.OrdersDB.InsertRecord "Products", rec
   If App.OrdersDB.Error then
    App.DisplayDatabaseError True
    Return
  End if
```

The following section is executed when the Part Number field contains a unique value. This means that the product already exists and it is necessary to update the record.

Listing 32. SaveProduct.Action, part 5.

```
else
// The part number field is read-only which means that this product
// already exists in the database, so we want to update it.

// Find the existing product record
  rs = App.OrdersDB.SQLSelect("SELECT * FROM Products WHERE " +_
      "PartNumber = '" + EscapeSQLData(productPartNumber.text) + "'")
  If App.ordersDB.error then
    App.displayDatabaseError False
    Return
  End if

  // Update the existing database record.  First we need to make the
  // RecordSet editable, which will try to obtain a lock on the record.
  // Be sure to check for errors after calling RecordSet.Edit in case
  // something went wrong here.
  rs.Edit
  If App.ordersDB.error then
    App.displayDatabaseError True
    Return
  End if

  // Update the data on the product record
  rs.Field("PartNumber").StringValue = ProductPartNumber.Text
  rs.Field("Name").StringValue = ProductName.Text
  rs.Field("Price").DoubleValue = moneyToDouble( productPrice.text )
  rs.Field("Image").StringValue = imageData

  // Update the record in the database
  rs.Update

End if

// Commit changes to the database
App.OrdersDB.Commit

// Update the list of products
UpdateProductList
```

Updating Database Records

Just as with adding records, there are also two ways of updating a record in a database table. For example if you need to update the number of products ordered, a SQL statement such as:

```
UPDATE LineItems SET Quantity = 7 WHERE LineItemID = 187
```

can be used. The more common technique, however, is to use the RecordSet class.

To update an existing database record, you first use a SQL SELECT statement to find the record. Then you call the Edit method of the RecordSet class and make changes to the record with the Field and IdxField methods of the RecordSet class. Then you save the changes using the RecordSet's Update method.

When using a multi-user database back-end such as REAL SQL Server, the Edit method of the RecordSet class locks the record so that another user cannot edit it. The Update method will save changes to the record and unlock it. If the Database class instance's Error method returns True after calling Edit, the record is currently locked by another user.

Listing 33. Multiuser database record locking example.

```
Dim rs as RecordSet
rs = REALSQLServerdb.SQLSelect("SELECT * FROM Orders WHERE " +_
        "OrderNumber = 00012457")
If rs.EOF = False then

// Attempt to lock record for editing
  rs.Edit

  // Check for privileges if in a multi-user database
  If REALSQLServerdb.Error()
    MsgBox "The record is currently locked."
    Return
  End if

  // Edit the Record
  rs.Field("TimesWatched").IntegerValue = 143

  // Update the Record
  rs.Update // Unlocks Record
End if
```

Updating a Customer Record

The StatCustomerNumber StaticText field will be empty for a record that needs to be created. If it contains a customer ID, the record is being updated.

If the current record already exists, the first thing to do is get the customer's customer ID from the StatCustomerNumber field so that the record can be selected.

After finding the record, you call the Edit method and then update all of the fields in the record to the current values stored in the controls in the interface.

Listing 34. SaveCustomer.Action, Update code.

```
// Find the existing record by searching for the customer id
rs = App.OrdersDB.SQLSelect("SELECT * FROM Customers WHERE " +_
          "ID = "+ EscapeSQLData(StatCustomerNumber.Text) + "'")
If App.ordersDB.error then
  App.displayDatabaseError true
  Return
End if

rs.Edit  //call Edit prior to modifiying the record and check for errors
If App.ordersDB.error then
  App.DisplayDatabaseError True
End if

// Update the customer information
rs.Field("Company").StringValue = CustomerCompany.Text
rs.Field("FirstName").StringValue = CustomerFirst.Text
rs.Field("LastName").StringValue = CustomerLast.Text
rs.Field("Phone").StringValue = CustomerPhone.Text
rs.Field("Fax").StringValue = CustomerFax.Text
rs.Field("Email").StringValue = CustomerEmail.Text
rs.Field("Address").StringValue = CustomerAddress.Text
rs.Field("City").StringValue = CustomerCity.Text
rs.Field("State").StringValue = CustomerState.Text
rs.Field("PostalCode").StringValue = CustomerPostalCode.Text

// Update the record in the database
rs.Update
if App.OrdersDB.Error then  // handle errors
  App.DisplayDatabaseError True
  Return
End if

// Commit changes to the database
App.OrdersDB.Commit

// Update the list of customers
UpdateCustomerList
```

Since the REAL SQL Database is single-user, it's not necessary to check whether the current record of the RecordSet is already locked by another user. However, there is a small chance that Edit may fail even in single-user, so you should check for errors before proceeding.

It is always necessary to call the Update method of the RecordSet class after making changes to the record so that the changes to the record will be saved to the database.

Updating a Product Record

Updating a product record is very similar to the updating a customer record. The only real difference is that the product's image has to be stored in a Blob column. The code to do this is shown in Listing 29 on page 40.

Updating an Order Record

Updating an order record is more involved than updating a product or a customer record because it needs to handle creating, modifying, and deleting line item records in addition to updating the order record itself. After each step you should check for errors, as in the example code.

If the StatOrderNumber field is not empty (i.e., it contains an order number), the selected record should be updated to contain the current data in the fields. The first step is to select the order using the order number to find the order record in the database.

Listing 35. SaveOrder.Action Update, part 1.

```
// We have an order number so we should update the existing order record.

// Get the order number of the selected Order
orderNumber = Val( StatOrderNumber.Text )

// Select the existing order record that we are updating
rs = App.OrdersDB.SQLSelect("SELECT * FROM Orders WHERE " +_
        " OrderNumber = '" + Str(orderNumber) + "'")

// Update the existing database record.  First we need to make the
// RecordSet editable which will try to obtain a lock on the record.
// Be sure to check for errors after calling RecordSet.Edit in case
// something went wrong here.
rs.Edit
If App.ordersDB.error then
  App.displayDatabaseError False
  Return
End if

// Update the database record
rs.Field("OrderNumber").StringValue = StatOrderNumber.Text
rs.Field("PurchaseOrder").StringValue = OrderPurchaseOrder.Text
rs.Field("CustomerID").StringValue = OrderCustomerNumber.Text
rs.Field("DateOrdered").StringValue = OrderOrderedOn.Text
rs.Field("DateShipped").StringValue = OrderShippedOn.Text
rs.Field("ShipMethod").StringValue = OrderShipMethod.Text
rs.Field("TaxRate").DoubleValue = Val(OrderTaxRate.Text)
rs.Field("SubTotal").DoubleValue = subTotal
rs.Field("TotalTax").DoubleValue = taxTotal
rs.Field("Total").DoubleValue = total

// Update the order record in the database
rs.Update
If App.OrdersDB.Error then
  App.DisplayDatabaseError True
  Return
End if
```

The code in Listing 35 follows the same pattern as in the Customers and Products tables, but now it is necessary to update the line items for the order as well.

Updating the line items attached to an order does not simply involve using the Edit-Change-Update procedure because an entire line item can be removed or added when editing an order.

The situation is complicated by the fact that the line items listbox can potentially hold four types of items: line items that are being added, line items that are being modified, line items that are left unchanged, and line items that are marked for deletion. The SaveOrder.Action event handler needs to handle all of these possibilities.

When looping through the rows of the LineItemsList ListBox, the CellTag property is checked for a StringValue. The CellTag property of the first cell in each row of the line items list is the LineItem ID for that line item record. If the CellTag is an empty string, it means that row does not already exist in the database (it was added by the user with the interface controls) and needs to be created.

Listing 36. SaveOrder.Action Update, part 2.

```
// The order record has been updated so now we need to make all changes
// to the line items on this order.  We will loop through all of the line items
// and update the data or insert a new record where an existing one does
// not already exist.

// Loop through all of the line items
For i = 0 to LineItemsList.listCount-1

// Look for a LineItemID in the CellTag of the first column.
// If one is found then a line item record already exists in the database,
// but if we don't find one then we need to create a new line item record.
  lineItemID = LineItemsList.CellTag( i, 0 )

  // Update the existing line item when we have an ID
  If lineItemID <> "" then
    lineSet = App.OrdersDB.SQLSelect("SELECT * FROM LineItems " +_
         "WHERE ID = '" + lineItemID + "'")
    lineSet.Edit  // make the recordset editable
    If App.ordersDB.Error then
      App.displayDatabaseError True
      Return
    End if

// update the data in the record
  lineSet.Field("OrderNumber").StringValue = StatOrderNumber.Text
  lineSet.Field("PartNumber").StringValue = LineItemsList.Cell(i, 0)
  lineSet.Field("ProductName").StringValue = LineItemsList.Cell(i, 1)
  lineSet.Field("Quantity").StringValue = LineItemsList.Cell(i, 2)
  lineSet.Field("Price").StringValue = LineItemsList.Cell(i, 3)
  lineSet.Field("LineTotal").StringValue = LineItemsList.Cell(i, 4)

  // update the record in the database
  lineSet.Update
  If App.ordersDB.Error then  //handle errors
    App.displayDatabaseError True
    Return
  End if
```

If the LineItemID is blank we need to create a new line item record.

Listing 37. SaveOrder.Action Update, part 3.

```
    else  // Create a new line item record
    rec = New DatabaseRecord
    rec.Column("OrderNumber") = StatOrderNumber.Text
    rec.Column("PartNumber") = LineItemsList.Cell(i, 0)
    rec.Column("ProductName") = LineItemsList.Cell(i, 1)
    rec.IntegerColumn("Quantity") = Val(LineItemsList.Cell(i, 2))
    rec.DoubleColumn("Price") = Val(LineItemsList.Cell(i, 3))
    rec.DoubleColumn("LineTotal") = Val(LineItemsList.Cell(i, 4))

    //  Insert new record into the database
    App.OrdersDB.InsertRecord "LineItems", rec
    If App.ordersDB.Error then
      App.displayDatabaseError True
      Return
    End if

  End if
Next   //loop through all the line items.
```

While updating the record, the user can delete a selected line item in the Line Items listbox by clicking the Delete button in the Order GroupBox. This triggers the DeleteLineItem's Action event. It adds the ID of the line item to the RemovedLineItems array. This array is referenced in the last section of the SaveOrder Action event handler, as it must delete these line items before saving the update.

Listing 38. The DeleteLineItem.Action.

```
Dim lineItemID as String
LineItemID = LineItemsList.CellTag(LineItemsList.ListIndex, 0)
If lineItemID <> "" then
  RemovedLineItems.Append lineItemID
End if

  // Remove the row from the listbox
LineItemsList.RemoveRow LineItemsList.ListIndex

// Recalculate the order totals
CalculateSubTotalAndTax()
```

If the rowID retrieved from the CellTag property is not empty, then the line item record is retrieved from the database using the LineItemID of the record. The Edit-Change-Update technique is then used to update the line item record with the values in the row of LineItemsList.

So far, all line item records that are either new or needed to be updated have been handled. To remove line items from the database which were removed from the line items list, the RemovedLineItems array is referred to. The RemovedLineItems array is an array of strings which contains the LineItemIDs of the line items that were removed from the list. This array is a window property that is maintained by the Delete-LineItem button in the Detail View, as shown below.

Listing 39. SaveOrder.Action, part 4.

```
// Now we need to delete any line item records that were marked
// to be removed.
// When a line item is deleted the record id is stored in an array, which we
// will loop through and delete all line items that we find in that array.

For Each lineItemID in RemovedLineItems
  App.OrdersDB.SQLExecute("DELETE FROM LineItems WHERE ID = '" +_
          lineItemID + "'")
  If App.OrdersDB.Error then// handle errors
    App.DisplayDatabaseError True
    Return
  End if
Next
```

When a user selects a row in the LineItemsList and clicks the DeleteLineItem button, the CellTag property of the first cell in the selected row is checked to see if it contains a LineItemID value. If it does, the value is appended to the RemovedLineItems array. This allows us to know which line items for the order should be deleted when the Save button is clicked.

A simple loop in the SaveOrder button's update code then goes through each of these IDs and then removes the line items from the database using a simple SQL statement.

Listing 40. Delete Rows loop in the SaveOrder.Action event.

```
// Now we need to delete any line item records that were marked
// to be removed.
// When a line item is deleted the record id is stored in an array, which we
// will loop through and delete all line items that we find in that array.

For Each lineItemID in removedLineItems
  App.OrdersDB.SQLExecute("DELETE FROM LineItems WHERE ID = '" +_
           lineItemID + "'")
  If App.OrdersDB.Error then // handle errors
    App.DisplayDatabaseError True
    Return
  End if
Next
```

The final block of code commits all the changes to the database and updates the List View.

```
// Commit changes to database
App.OrdersDB.Commit

// Update the list of orders
UpdateOrderList
```

Deleting Database Records

Just as with adding and updating records, there are two ways of deleting a record in a database table. A SQL statement (like the one to remove the old line items while updating an order) or the Delete method of the RecordSet class can be used.

To use a RecordSet instance to delete a record, you first select the record you want to delete using a SELECT statement, then, assuming the record was found, call the Delete method of the RecordSet to delete it. This should then be followed up by a call to the Commit method for the database instance to permanently save the change.

Listing 41. Deleting via the RecordSet.Delete method.

```
Dim rs as RecordSet
rs = App.OrdersDB.SQLSelect("SELECT * FROM Customers WHERE" +_
        "CustomerNumber = '00218974'")
rs.Delete
App.OrdersDB.Commit
```

Deleting a Customer Record

When deleting a customer, *all* of the information related to that customer will be deleted as well. This means that all orders for that customer, and all of the line items for those orders have to be deleted. This may sound difficult, but it’s really quite simple.

Before deleting the customer, a confirmation dialog will appear to warn the user of the permanency of the action. If the customer does not confirm the deletion, the operation is aborted. This is the first section of code.

Listing 42. DeleteCustomer.Action, part 1.

```
// Delete the selected customer record and all other records that are
// related to that customer.

Dim customerID, orderNumber as String
Dim rs as RecordSet

// Ask the user to confirm the delete
If Not ConfirmDelete("customer") then
// The delete was not confirmed so we are going to return out of this
// event so that the delete does not take place.
  Return
End if
```

ConfirmDelete is a simple method that displays a MessageDialog box. It is passed the type of item that is being deleted and displays the passed string in the MessageDialog.

Listing 43. The ConfirmDelete method.

```
Function ConfirmDelete(Section as String) as Boolean
// Display a dialog that asks the user to confirm their delete action.
// Return True if the delete has been confirmed.

  Dim dlog as MessageDialog

// Create the Dialog
  dlog = New MessageDialog
  dlog.Message = "Are you sure you want to delete the selected " + section + "?"
  dlog.Explanation = "This action cannot be undone."
  dlog.ActionButton.Caption = "Delete"
  dlog.CancelButton.Caption = "Cancel"
  dlog.CancelButton.Visible = True

// Show the Dialog
// Return True if confirmed
  If dlog.ShowModal.Caption = "Delete" then
    Return True
  End if

// Cancel the delete
  Return False
```

In the DeleteCustomer Action event handler, the first task is to select all of the Order-Numbers from the orders the customer has placed. If there are no orders, the program continues on to delete the customer record from the Customers table by using the

SQL DELETE command. If there orders for the customer, a SQL DELETE statement is used to delete the line items for the order, and then the order itself.

Listing 44. DeleteCustomer.Action, part 2.

```
// Start a transaction so that we can assure that all of the records
// have been deleted and we cannot get into a situation where we have a
// partial delete.

App.ordersDB.sqlExecute "begin transaction"

// Get the selected Customer's ID which is stored in the first column
// of the listbox
customerID = CustomersList.Cell( CustomersList.ListIndex, 0 )

// First we want to delete all of the customer's orders.
// An order has LineItems attached to it so we will select the orders for this
// customer and delete the line item records for each of those orders as well.

rs = App.OrdersDB.SQLSelect("SELECT OrderNumber FROM Orders " +_
            "WHERE CustomerID = '" + customerID + "'")
While rs.EOF = False
  // get the order number for this order
  orderNumber = rs.IdxField(1).StringValue

  // delete all of the line items for this order
  App.OrdersDB.SQLExecute("DELETE FROM LineItems WHERE " +_
          OrderNumber = '" + orderNumber + "'")
  If App.OrdersDB.Error then        // handle errors
    App.DisplayDatabaseError True
    Return
  End if

  // delete the order record
  App.OrdersDB.SQLExecute("DELETE FROM Orders WHERE " +_
          OrderNumber = '" + orderNumber + "'")
  If App.OrdersDB.Error then      // handle errors
    App.DisplayDatabaseError true
    Return
  End if

  // move to the next order
  rs.MoveNext
Wend
rs.Close
```

After deleting all of the records and handling any errors, the changes are committed to the database, and the List View is updated.

Listing 45. DeleteCustomer.Action, part 3.

```
// Now delete the actual customer record
App.OrdersDB.SQLExecute("DELETE FROM Customers WHERE ID = '" +_
         customerID + "'")
If App.OrdersDB.Error then   // handle errors
  App.DisplayDatabaseError True
  Return
End if

// Commit changes in the transaction
App.OrdersDB.Commit

// Update the list of customers
UpdateCustomerList
```

Deleting Product and Order Records

Deleting product and order records is nearly identical to deleting customer records. The only differences are that the part and order numbers are used to find the records to be deleted.

Sharing Data

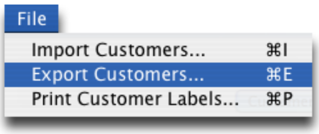
Since importing, exporting, and printing data from a database is very often a requirement for an application designed around the management of data, the Orders database provides examples of importing and exporting customer records, and printing shipping labels for customers.

Exporting Data

Data formats are as varied as the applications which use them. Very often the question of a new programmer is how to export data from their applications, usually because they are unsure of how to do it or the format to do it in.

This example uses a standard format, a tab-delimited text file with fields in a record separated by tabs and the records separated by EndOfLine characters. This is an extremely simple format to read and write in any language (especially REALbasic), and best of all, it is also easily read and edited by the user.

Figure 13. The Export Customers menu item is used to export customers in the Orders database example.



Exporting takes place in the menu handler for the File ▶ Export Customers menu item. An instance of the SaveAsDialog class will appear asking the user for the location to export the data to. With that result, the file is then created using a TextOutputStream. If an error occurs, it is handled accordingly.

Listing 46. FileExportCustomers MenuHandler, part 1.

```
// Export all of the customer data to a tab delimited text file.

Dim dlog as SaveAsDialog
Dim file as FolderItem

// Create a save dialog and ask the user to select a location for the
// exported file to be saved.
dlog = New SaveAsDialog
dlog.PromptText = "Export customers to"
dlog.SuggestedFileName = "Customers.txt"
file = dlog.ShowModalWithin(Self)

// If the user cancelled then we do not need to proceed
If file = NIL then
  Return True
End if

// Declare variables for exporting the customer data
Dim tout as TextOutputStream
Dim customer as String
Dim rs as RecordSet

// Create the text file
tout = file.CreateTextFile()
If tout = Nil then
  Beep
  MsgBox "Export Error" + EndOfLine + EndOfLine + _
           "Could not create the customers file."
  Return True
End if
```

Next, a standard SQL SELECT statement is used to retrieve all the records from the Customers table, and a local string variable, “customer”, is assigned the concatenated values of all the fields in the record. A tab character (ASCII code 9) is placed between fields and an EndOfLine character is placed between the records using the TextOutputStream’s WriteLine method.

Listing 47. FileExportCustomers MenuHandler, part 2.

```
// Select all customer data from the database
rs = App.OrdersDB.SQLSelect("SELECT * FROM Customers")
if App.ordersDB.error then
  App.displayDatabaseError False
  Return True
End if

// Loop through all of the records and write the data out to a file
While Not rs.EOF

  // Build a string for this record which will put a tab between each
  // column of data.
  customer = rs.Field("Company").StringValue
  customer = customer + chr(9) + rs.Field("FirstName").StringValue
  customer = customer + chr(9) + rs.Field("LastName").StringValue
  customer = customer + chr(9) + rs.Field("Phone").StringValue
  customer = customer + chr(9) + rs.Field("Fax").StringValue
  customer = customer + chr(9) + rs.Field("Email").StringValue
  customer = customer + chr(9) + rs.Field("Address").StringValue
  customer = customer + chr(9) + rs.Field("City").StringValue
  customer = customer + chr(9) + rs.Field("State").StringValue
  customer = customer + chr(9) + rs.Field("PostalCode").StringValue

  // write out the line to the text file
  tout.WriteLine customer

   // move to the next record
  rs.MoveNext
Wend

// close the recordset
rs.Close

// close the file
tout.Close

// we handled this menu item
Return True
```

Importing Data

In this example, customer records can only be imported in the same format that is used when exporting from the database. Each customer record must be on its own line, and each field is separated by tabs. The fields must be in the same order as in the export.

Importing customer records from a file is just as easy as exporting them. After a dialog confirms the file to import, it is opened with a TextInputStream.

Listing 48. FileImportCustomers MenuHandler, part 1.

```
// Import customer data from a text file.  This will read the data in from
// a tab-delimited text file and create new customer records with that data.

Dim dlog as OpenDialog
Dim file as FolderItem

// Create an open dialog to allow the user to select the file to import
dlog = New OpenDialog
dlog.PromptText = "Select a Customers file to import"
dlog.Filter = "text"
file = dlog.ShowModalWithin(Self)

// If the user cancelled then the file will be Nil.  Since we didn't select
// a file we don't have anything to do so we can just exit out.
If file = NIL then
  Return True
End if

// Import the customer data from the text file
Dim tin as TextInputStream
Dim customer as String
Dim rec as DatabaseRecord

// Open the text file
tin = file.OpenAsTextFile
If tin = Nil then
  Beep
  MsgBox "Import Error" + EndOfLine + EndOfLine + _
            "There was an error when opening the customers file."
  Return True
End if
```

The next block of code uses SQL's transaction feature to enclose the import code in a transaction. This will protect the database against partial or failed imports. The database will not actually be affected until the changes are committed at the end of the transaction.

A While-Wend loop in conjunction with the ReadLine method of TextInputStream is used to continuously read records (since each line of the file is a record) from the file until the end is reached. ReadLine returns a customer record whose fields are separated by tabs. The NthField function retrieves the individual fields from the single line of text. The fields are then placed into a newly created DatabaseRecord instance, and then inserted into the Customers table in the database.

Listing 49. FileImportCustomers MenuHandler, part 2.

```
// Start a new transaction.  All actions that are done inside of a transaction
// must be commited when finished in order to actually affect the database,
// but you also have the option of rolling back the changes in the
// transaction.  We are going to use a transaction during the import so that
// if any of the records fail to import, the data in the database will not be
// changed.  This prevents us from getting into a situation where we have
// only partially imported the file.
App.ordersDB.SQLExecute "begin transaction"
// read the file line by line
While not tin.EOF
  // Get the next customer record from the file
  customer = tin.ReadLine
  // If the line is blank then we can move on to the next line
  If customer = "" then
    Continue
  End if

  // Create a new customer record from this line.  We will use the
  // NthField function to pull data from between the tab characters.
  rec = New DatabaseRecord
  rec.Column("Company") = NthField(customer, chr(9), 1)
  rec.Column("FirstName") = NthField(customer, chr(9), 2)
  rec.Column("LastName") = NthField(customer, chr(9), 3)
  rec.Column("Phone") = NthField(customer, chr(9), 4)
  rec.Column("Fax") = NthField(customer, chr(9), 5)
  rec.Column("Email") = NthField(customer, chr(9), 6)
  rec.Column("Address") = NthField(customer, chr(9), 7)
  rec.Column("City") = NthField(customer, chr(9), 8)
  rec.Column("State") = NthField(customer, chr(9), 9)
  rec.Column("PostalCode") = NthField(customer, chr(9), 10)

  // Insert the record into the database
  App.OrdersDB.InsertRecord "Customers", rec

  // Error Handling
  If App.OrdersDB.Error then
    App.DisplayDatabaseError true
    Return True
  End if
Wend
```

After all of the records from the file have been inserted into the database, the changes are committed to the database, completing the transaction. Because new data has been imported, the ListBox for the Customers tab is repopulated.

Listing 50. FileImportCustomers MenuHandler, part 3.

```
// Close the text file
tin.Close

// Commit the changes that happened inside of the transaction
App.OrdersDB.Commit

// Update the list of customers
UpdateCustomerList

// we handled this menu action
Return True
```

Printing Data

Printing is more involved than writing to or reading from a text file, but it should be easy for anyone who has worked with and drawn in a Canvas control. The central task in printing is simply "drawing" onto a page with the Graphics class methods.

Figure 14 is some example output from the method that prints customer mailing labels.

Figure 14. An example of a printed page of customer labels.

The first step in printing is to open the printer settings dialog using the OpenPrinter-Dialog function. It returns a Graphics object into which we draw the text to be printed.

Listing 51. FilePrintCustomerLabels MenuHandler, part 1.

```
// Print customer labels.  When you print in REALbasic you are provided
// with a graphics class instance from the printer which represents a sheet
// of paper.  You then draw the contents to the paper (graphics class) and
// send it to the printer.

Dim g as Graphics
Dim rs as RecordSet
Dim customer as String
Dim columnIndex, columnWidth as integer
Dim rowOffset, rowHeight as Integer

// First we need to get a graphics class from the printer.  We will use the
// OpenPrinterDialog function to bring up the standard print window and
// allow the user to select their printer and print options.

g = OpenPrinterDialog()

// If the user clicked cancel in the print dialog then the resulting graphics
// class will be NIL.
If g = NIL then
  Return True
End if

// Set the text font and size that we want to print with
g.TextFont = "Arial"
g.TextSize = 9
```

The layout of the printed page in this example will be three columns of customer information, using as many rows as can fit on the page. The number of columns can be changed by changing the value of the constant “kColumnCount”. The customer’s information will consist of the customer’s company (if one was given), the customer’s name, and the address.

After setting the text size settings, the width of each column is determined by dividing the Width property of the Graphics object (the width of the printable area on the page) by the number of columns. The height of a row is set to be the height of a string with four lines of text.

Listing 52. FilePrintCustomerLabels MenuHandler, part 2.

```
// We are going to print the customer labels in columns so we need to
// calculate how wide each column is.  We can determine the width of each
// column by dividing the entire width of the page by the number of
// columns.  We are going to use 3 columns for this page.

const kColumnCount = 3
columnWidth = g.width / kColumnCount

// Calculate the height of a single record.
// Since each record contains 4 lines of data we will
// take the height of a line and multiply it by 4.
rowHeight = g.textHeight * 4

// Select all of the customer data from the database
rs = App.OrdersDB.SQLSelect("SELECT Company, FirstName, LastName," +_
         "Address, City, State, PostalCode FROM Customers")
If App.ordersDB.Error then
  App.DisplayDatabaseError False
  Return True
End if
```

After selecting all of the customer records from the table, the while loop will loop as long as there are more customers in the RecordSet “rs” to print. The inner for-loop goes through records three at a time, to print them on a row. If the end of the Record-Set has not been reached (meaning all customers have been printed), a string is set to the customer information and then drawn into the graphics context using the column index value from the for loop multiplied by the width of a column (which is the horizontal offset from the left side of the printable area on the page), as well as the vertical offset of the row as the coordinates to the DrawString method.

Listing 53. FilePrintCustomerLabels MenuHandler, part 3.

```
 // Loop through all of the records and draw them to the page to be printed.
While not rs.eof

  // Draw the next 3 records into the columns
  For columnIndex = 0 to kColumnCount-1
    // if we have reached the end of the recordset then there are not
    // any more customer records to print
    If rs.eof then
      Exit
    End if

    // Format the customer data into a label
    customer = rs.Field("Company").StringValue + EndOfLine
    customer = customer + rs.Field("FirstName").StringValue + " " +_
          rs.Field("LastName").StringValue + EndOfLine
    customer = customer + rs.Field("Address").StringValue + EndOfLine
    customer = customer + rs.Field("City").StringValue + ", " +_
          rs.Field("State").StringValue
    customer = customer + " " + rs.Field("PostalCode").StringValue

    // Make all of the customer data uppercase
    customer = Uppercase(customer)

    // Draw the customer data to the page in the appropriate column
    // and row
    g.DrawString customer, columnIndex * columnWidth, rowOffset +_
           g.TextAscent

    // move to the next customer
    rs.MoveNext
  Next

  // If we don't have any more records then we are done printing
  If rs.eof then
    Exit
  End if
```

After every three addresses are printed, the vertical offset for the next row is calculated by adding the row height to the previous offset. To determine if the next row will fit on the page when printed, the offset is checked to be greater than the Height value of the Graphics object. If it is, the row will not fit and must be printed on the next page. Calling the NextPage method of the Graphics object sends the current page to the printer and clears the Graphics object so that the next page can be drawn. It's important to note that the page is not actually printed until either the NextPage method is called or the Graphics object goes out of scope and is destroyed.

Listing 54. FilePrintCustomerLabels MenuHandler, part 4

```
// We finished printing to that row so now we need to move down
// to the next row for the next columns to be drawn.
// Move down the height of one row and then add in the height
// of a line so that there is a blank line between the addresses.

  rowOffset = rowOffset + rowHeight + g.textHeight

// If we have moved down to the end of the page then we need to send
//this page to the printer and request a new blank one.

  If rowOffset > g.Height then
    // Send page to the printer and clear out graphics class for next page
    g.NextPage

    // Move back to the top row of the page
    rowOffset = 0
  End if
Wend

// The last page is sent to the printer when the graphics class goes
// out of scope at the end of this method.
Return True   // We handled this menu action
```

Index of Code Examples