BASIC PROGRAMMING

# Gambas Easy database access

**PART 1** Do you need to build a nice GUI front-end for your database? Have you only got ten minutes to do it? **Dr Mark Alexander Bain** prescribes Gambas.

**ON the DISC**

- **Gambas 1.0.8 (stable)**
- **Gambas 1.09 (development)**
- **Project code from the tutorial**

I wonder if I can say this without you throwing a wobbly – how would you like Visual Basic on Linux? Now, just try to calm down, remember to breathe, and please, please stop tearing up the magazine. How about instead, I said there are a lot of very experienced Visual Basic programmers out there who would like to fully commit to Linux, but who won't move until they are able to bring all of their skills with them? Suppose there were a language that wasn't VB, but allowed VB programmers to move to Linux. Now that sounds better doesn't it?

Gambas is that language. It's designed so that anyone experienced in using Visual Basic will feel completely at home. Equally, someone who has never even thought of programming before will be able to produce a professional-looking application.

By the end of this tutorial you will be able to build a GUI, set up a (simple) database and be able to use the GUI to read and write to the database you've just created.

Right – if it's so easy let's get on with it!

## First things first

Time to install the database. I know you just want get on to the fun bit – creating the GUI and getting into the coding – but it is essential to make sure that the database is in place. Why? Well, the installation process for Gambas needs the drivers that come with the database. It will compile without them and you'll be able to use it, but it won't enable the support for the database.

Now for the database itself. If you immediately think of *MS Access*, wash your mind out with soap and water. I assume that you will be using either *MySQL* or *PostgreSQL*. Why? They are both freely available, easy to use and very stable.

The main disadvantage to using *PostgreSQL* is that Gambas expects you to have the *PostgreSQL* development packages installed (for the drivers). This means that you will need to hunt down the files for your particular flavour of Linux. If you are unsure, or if you are in a hurry (or if you're just plain lazy like me), *MySQL* is the one to go for. Gambas is able to work with it straight out of the box.

There is nothing in the Gambas installation process that you wouldn't expect, and you'll soon be able to run the application. Download Gambas from **http://gambas.sourceforge.net**. The first screen that you'll see is the Gambas welcome screen, and from here you can create a new project or open existing ones.

**QUICK TIP**

**Examples**
**If you want to learn as much as you can about Gambas, click on Examples on the welcome screen. In here you'll find a sample of most operations that you could want to do.**

Clicking on New Project will bring up a wizard to guide you through the setup of your project. Once the project is open, the first thing to do is to create a new form (this is a GUI after all!). Go to the Project window and click on the Forms folder. Click the right mouse button and then New.

## Building up the GUI

You can now start to build the GUI by clicking on a required object in the toolbox, then using the mouse to draw it on to the form. For this example we will just add a combo box and a text box. Click on the green triangle to run the application. It won't do much at the moment, but we can now add code to do more interesting work. To go back to design mode click on the red box in the menu of the project window.

Gambas is event-driven (just like Visual Basic). This means that all the coding is associated with buttons, combo boxes and so on. If you double click in the body of the form, the application will take you into the coding area. At the moment it will look something like:

```
' Gambas class file
PUBLIC SUB Form_Open()
END
```

This is the area in which we can place variables, subroutines, functions and comments (denoted by **'**). By default the system creates the **Form_Open** subroutine, that is, the subroutine that runs in the event of the form opening. That's what makes Gambas event-driven.

Now modify the code so that it reads:

```
' Gambas class file
PUBLIC SUB Form_Open()
        combobox1.Add("Fred Jones")
        combobox1.Add("Mary Smith")
        combobox1.Add("Jim Thompson")
END
```

Click on the green Run button in the Project window. You'll now have a form with a working combo box. Exciting or what?

Put the form back into design mode (by clicking on the red Stop button) and double-click on the combo box. The application will create a new subroutine (**ComboBox1_Click**) for you that will look like this:

```
' Gambas class file
PUBLIC SUB Form_Open()
```

```
  combobox1.Add("Fred Jones")
  combobox1.Add("Mary Smith")
  combobox1.Add("Jim Thompson")
  ComboBox1_Click
END
PUBLIC SUB ComboBox1_Click()
  IF combobox1.text = "Fred Jones" THEN
  textbox1.Text="London"
  ELSE IF combobox1.text = "Mary Smith" THEN
  textbox1.Text="Paris"
  ELSE IF combobox1.text = "Jim Thompson" THEN
  textbox1.Text="New York"
  END IF
END
```

Notice that the subroutine **ComboBox1_Click** is called at the end of **Form_Open**. Try running the app and then re-running it with the subroutine commented out to see why it's included.

We now have a working GUI, but all the details such as names are hard coded into the application – not the best of situations. Imagine distributing this throughout a company and then finding out that Fred is actually in Berlin, or Mary has just got married (or divorced), or even that Jim is now Jane. By using a database as the main store for all of the information we will be able to build an application that is much easier to maintain.

## Setting up the data

The database that you've selected will need some preparation before the application is ready.

The code for setting up the initial database environment varies, depending on whether you use *MySQL* or *PostgreSQL*.

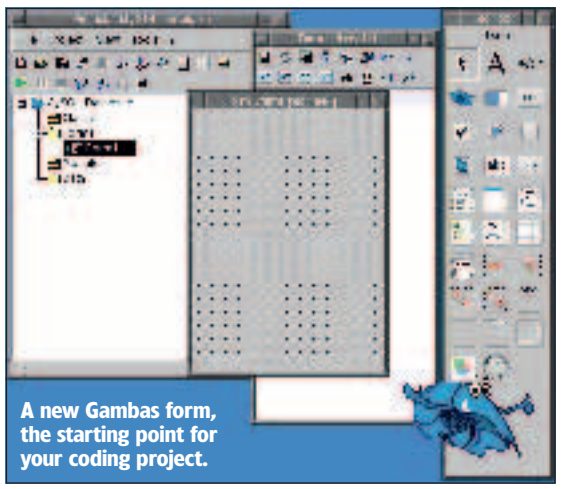
| MySQL (log on as root) | PostgreSQL (log on as root) |
|---|---|
| | useradd -d /home/postgres postgres |
| | mkdir /home/postgres |
| | chown postgres /home/postgres |
| su mysql | mkdir /usr/local/pgsql/data |
| mysql_install_db | chown postgres /usr/local/pgsql/data |
| | su postgres |
| | /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data |

This code only needs to be run once, immediately after you've installed the database software.

To start the database, run

| MySQL (log on as mysql) | PostgreSQL (log on as postgres) |
|---|---|
| /usr/bin/mysqld_safe & | /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data >~/logfile 2>&1 & |

Obviously this will need to be run every time that you restart your computer.



A new Gambas form, the starting point for your coding project.

| MySQL (log on as any user) | PostgreSQL (log on as postgres) |
|---|---|
| echo "create database customers"\| mysql -uroot | /usr/local/pgsql/bin/createdb customers |
| mysql -uroot < data.sql | /usr/local/pgsql/bin/psql customers < data.sql |
| create table manager ( | create table manager ( |
| id int auto_increment, | id bigserial, |
| surname varchar(50), | surname varchar(50), |
| firstname varchar(50), | firstname varchar(50), |
| primary key (id)); | primary key (id)); |
| create table office ( | create table office ( |
| id int auto_increment, | id bigserial, |
| city varchar(50), | city varchar(50), |
| manager_id int, | manager_id int, |
| primary key (id)); | primary key (id)); |
| GRANT select,insert,delete,update ON * | create user bainm password 'mypassword'; |
| TO bainm@localhost IDENTIFIED BY 'mypassword'; | GRANT select,insert,delete,update on manager to bainm; |
| | GRANT select,insert,delete,update on office to bainm; |

The code to load data into the table is:

```
/*Load default data*/
insert into manager (firstname,surname) values
('Fred','Jones');
insert into manager (firstname,surname) values
('Mary','Smith');
insert into manager (firstname,surname) values ('Jim',
'Johnson');
insert into office(city,manager_id) values ('London',1);
insert into office(city,manager_id) values ('Paris',2);
insert into office(city,manager_id) values ('New York',3);
```

Although the tools for each database vary, the basic language is pretty well the same. Both programs use SQL (Structured Query Language), but there are some differences in the way that SQL is implemented. For example, in the SQL shown above *MySQL* allows an asterisk, *, to be used in the **GRANT** statement, but *PostgreSQL* expects a separate **GRANT** for each table used. However, the use of SQL does mean that in most cases you can safely write a query that will work with either database.

Before we can send queries to the database we need to connect to it. Gambas contains a number of components that can be added at design time; in this case a database component needs to be added. Click on Project in the Project Window menu, and then Properties. Go to the Components tab and place a tick next to **gb.db**. The application is now ready for connection to a database.

The actual connection is done through coding, but is very simple coding. First the application must be told that there is a connection to be used:

```
' Gambas class file
PRIVATE conn AS NEW Connection
```

Place this top of the code file for the form, outside of any functions or subroutines. This makes the connection available everywhere in the form as a global parameter. Having done this we can create a subroutine that makes the actual connection.

The first thing to do is to give to the connection the details that it must use: the login details, the database name and location, and the driver to be used.

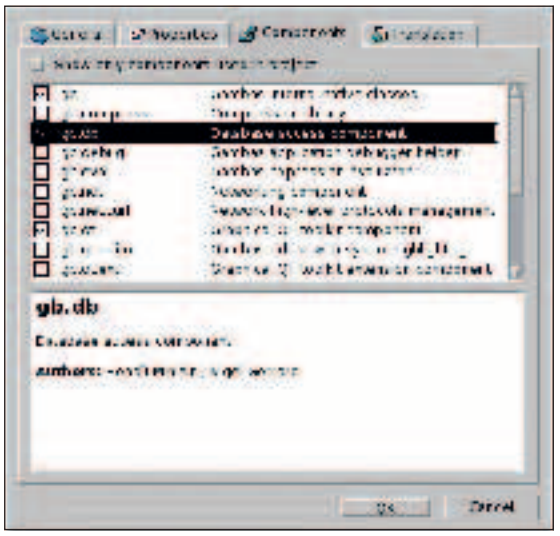
```
WITH conn
                .Type = "mysql"
                .Host = "localhost"
                .Login = "bainm"
                .Password = "mypassword"
                .Name = "customers"
END WITH
```

Having told the connection what to do, we can then tell it to open so that it is ready for retrieving or sending data:

```
TRY conn.OPEN
```

### THE GAMBAS MASCOT

**Please don't be put off by the big blue prawn – the Gambas mascot. Happily, as well as being the recursive acronym for Gambas almost means basic, the word *gambas* also means shrimp in Spanish. Olé! If you find it particularly irritating go to the Project Window, click on Tools, then Preferences.... Go to the Others tab and you'll be able to turn the mascot off.**



**The Gambas Project Properties screen, where we can link the application to a database.**

We use the **TRY** statement because we don't want the application just to crash if it can't connect; instead we want to catch and handle any errors, with the following:

```
TRY conn.Open
IF ERROR THEN
        Message ("Cannot Open Database. Error = " &
Error.Text)
END IF
```

The complete function for making connections would be this:

```
' Gambas class file
PRIVATE conn AS NEW Connection
PRIVATE FUNCTION make_connection() AS Boolean
        WITH conn
                .Type = "mysql"
                .Host = "localhost"
                .Login = "bainm"
                .Password = "mypassword"
                .Name = "customers"
        END WITH
        TRY conn.Open
        IF ERROR THEN
                Message ("Cannot Open Database. Error
= " & Error.Text)
                RETURN FALSE
        END IF
        RETURN TRUE
END
```

The function **make_connection** will return true if the connection can be made, and false if it can't.

## Time to use the data

Having connected to the database we want to be able to make use of the data that it stores – we're going to use it to populate the combo box.

Any Visual Basic programmer will be used to the concept of the RecordSet. In Gambas this is known as the Result. Either way, it's a container for any information returned to the application as the result of any queries run against the database. For instance, if we wanted a list of all of the managers stored in the database we would use this query:

```
select surname,firstname from manager
```

We can send this query to the database (via the connection that we've set up), and load the resulting data into a Result.

```
resManager= conn.Exec("select surname,firstname from
manager")
```

In this instance the information will be used to populate the combo box. To do this we can use a simple loop:

```
FOR EACH resManager
  ComboBox1.Add (resManager!firstname & " " &
resManager!surname)
 NEXT
```

Notice that each field is referenced using the Result name, an exclamation mark and the field name, as in **res!firstname**.

If **resManager** is made global it will be available to other subroutines and functions, and including the **id** field will enable us to reference the data easily from within the program (more on that in a minute). Note how **&** is used to join statements.

```
PRIVATE resManager AS Result
PRIVATE SUB load_combo()
 DIM sql AS String
 sql="select id, surname,firstname from manager"
 resManager= conn.Exec(sql)
 FOR EACH resManager
  ComboBox1.Add (resManager!firstname & " " &
resManager!surname)
 NEXT
END
```

We can now populate the combo box with information directly from the database. The big advantage here is that any changes made to the data are automatically passed through to all users of the system.

Next we can use the data from the combo box to obtain more information from the database. In our example it will be the office where each manager is based. The idea is much the same as loading the combo box: a query will be sent to the database, and the returned results will be used to populate a text box. The main difference is that the returned data will depend on the contents of the combo box.

Back in the subroutine **load_combo**, we loaded the manager list into **resManager**. This stores the ID, first name and surname of each manager. All we have to do is move to the record, look up the ID, and query the database to get the office location according to the manager's ID. We can use the combo box index to identify which record to go to (1st manager = index 0, 2nd = 1, 3rd = 2, etc). The technique is simple: move to the first record, then move *n* places, where *n* is the index number.

```
resManager.MoveFirst
resManager.MoveTo( ComboBox1.Index)
```

The SQL can be created using **resManager!id**:

```
sql="select city from office" &
 " where manager_id=" & resManager!id
```

The final subroutine will, therefore be:

```
PUBLIC SUB ComboBox1_Click()
        DIM res AS Result
        DIM sql AS String
        resManager.MoveFirst
        resManager.MoveTo( ComboBox1.Index)
        sql = "select city from office" &
        " where manager_id=" & resManager!id
        res = conn.Exec(sql)
        textbox1.Text=res!city
END
```

Finally **Form_Open** needs to be updated:

```
PUBLIC SUB Form_Open()
        IF make_connection()=TRUE THEN
                load_combo
                ComboBox1_Click
        END IF
END
```

We've already done some writing to the database when the database was set up. That was using an **INSERT** statement. The other way to write to the database is to use an **UPDATE** statement. Let's look at **INSERT** first.

### QUICK TIP

**Query with care**

**Take care when querying the database as it will have an effect on the performance of your application. Loading all of your data when the application starts will slow down the startup but minimise the network traffic. Constantly querying the database will ensure all data is up to date, but will increase the load on the network and the database. So look at your data carefully, and only call the database when it's absolutely necessary.**

For this example we'll add a text box and a button. We'll then use them to insert a new city. Double-click on the button and you should find yourself in the **Button1_Click** subroutine:

```
PUBLIC SUB Button1_Click()
END
```

The general process that the subroutine is going to follow is to create the SQL and then send the query to the database:

```
sql="insert into office (city) values ('" & textbox2.text &"')"
```

In the actual subroutine we'll also add some checking to ensure that there is something typed into the box:

```
PUBLIC SUB Button1_Click()
  DIM res AS Result
  DIM sql AS String
  IF textbox2.text <> "" THEN
     sql="insert into office (city) values ('" & textbox2.text &"')"
     res=conn.Exec(sql)
  ELSE
     message ("Input required")
  END IF
END
```

You'll notice that the button displays **Button1,** and **TextBox2** displays **TextBox2**. To change these make sure that you're in design mode, click (only once because you don't want to write any code), and press F4. This will display the Properties page. Find the box entitled Text, change the contents to Add for the button and empty it for the textbox.

If you run the application and try to add another city nothing obvious will happen. However, if you go back to the command line and type

```
echo "select * from office"| mysql -uroot customers
```

you will find that the city has been added to the database.

To see the new cities within the application we can add another combo box, and populate this with all cities that do not have a manager. The form will need the new combo box added, and it will need code to populate it (which should run at **Form_Open**, and when the Add button is clicked).

```
PRIVATE resNewOffice AS Result 'Global variable
PRIVATE SUB load_new_office ()
        DIM sql AS String
        sql="select id,city from office where manager_id is NULL"
        resNewOffice=conn.Exec(sql)
        combobox2.Clear
        FOR EACH resNewOffice
                combobox2.Add(resNewOffice!city)
        NEXT
END
```

**The completed Gambas application.**

The function is very similar to the function for populating the manager combo box, with one main difference:

```
combobox2.Clear
```

This is because as well as being called from **Form_Load** (when the form opens) it will also be called whenever a new city is added. This line empties the combo box so that it can be repopulated with fresh data.

Finally, we can use the **UPDATE** query to change data in the database. In this case we're going to move one of the managers to a new office. This time the subroutine must move the manager id from the old office to the new one.

```
PUBLIC SUB Button2_Click()
        DIM res AS Result
        DIM sql AS String
        resNewOffice.MoveFirst
        resNewOffice.MoveTo (combobox2.Index)
        resManager.MoveFirst
        resManager.MoveTo (combobox1.Index)
        sql="update office set manager_id=NULL" &
        " where manager_id=" & resManager!id
        res=conn.Exec(sql)
        sql="update office set manager_id=" &
resManager!id &
        " where id=" & resNewOffice!id
        res=conn.Exec(sql)
        ComboBox1_Click
        load_new_office
END
```

All that remains is to tidy up the screen itself by adding labels (use F4 to access the Properties screen to change the text).

## Using stored procedures

The stored procedures are just like any other procedure or function, except that they're stored on the database instead of a client application. The advantage is that if you make any changes to the code all users will immediately see the changes (or the error codes) without having to redistribute the software. This can be very useful for the **UPDATE**/**INSERT** statements.

If you change the structure or location of a table, you can simply update the stored procedure on the database rather than having to rebuild and distribute the application. Creating the stored procedure (on *PostgreSQL* or *MySQL 5.0*):

```
mysql> CREATE PROCEDURE updateLocation (IN manid
INT,IN offid INT)
   -> BEGIN
   ->   update office set manager_id=NULL where manager_
id=manid;
        update office set manager_id=manid where id=offid;
   -> END
   -> //
```

And using the stored procedure (from within Gambas):

```
PUBLIC SUB Button2_Click()
   DIM res AS Result
   DIM sql AS String
   resNewOffice.MoveFirst
   resNewOffice.MoveTo (combobox2.Index)
   resManager.MoveFirst
   resManager.MoveTo (combobox1.Index)
   sql="call updateLocation(resManager!id,resNewOffice!id)"
   res=conn.Exec(sql)
   ComboBox1_Click
   load_new_office
END
```

Programming with Gambas is easy. If you are an experienced programmer stuck in Windows, perhaps you can see a way into Linux, thanks to the little blue Gambas. LXF

**NEXT MONTH**

**Next month we'll look at global parameters, modules and how to re-use functionality, and programming using classes.**