New features of PHP-GTK2

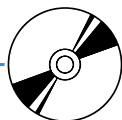
Pablo Dall'Oglio

PHP-GTK1 started a new way of thinking in PHP. The language devoted for web applications started to be widely used for writing the standalone, client-side applications with a Graphical User Interface (GUI). However, it is the release of PHP-GTK2, which is a binding between PHP5.1 and the Gtk-2.6 library, that can start a real revolution.

W SIECI

1. *http://gtk.php.net* – a homepage of the PHP-GTK project
2. *http://www.agata.org.br* – Agata Project – a CRM/ERP using PHP-GTK
3. *http:///tulip.solis.coop.br*

NA CD



Please don't translate the `underlined` text, unless it's marked in `yellow`.

We will take a closer look on the most outstanding features of this technology, writing the real life example: an application that allows simple management of items (products) in the store. Beside PHP-GTK2, it will make use of the new features of PHP5 (as the new object model and exception handling), and will store its data in the SQL database.

The concept

Let's think of what we'll need for our application. First of all, every GUI application has its main window, that contains other elements. In our case, those will consist of three pull-down menus named *File*, *Edit* and *Help*. All positions in our pulldown menus will have icons beside the text. In the first one, we will be able add a new item (*Add*), edit an existing one (*Edit*), clear the whole database (*Clear*) or quit the program (*Quit*). The second one will allow us to open a simple text editor, which may be useful for taking notes about our stock. The third one will contain a single option: *Help*, which will pop up an *About* Dialog. Being at the dialogs, we will also need (among others) the information and question dialogs for the *Clear* option from the first menu, a more complex dialog for adding new products and editing the existing ones, the file dialog and the independent window for our text editor.

What you should know...

You should be skilled in PHP5 (with focus on the new object model and exceptions) and have some knowledge of GUI building.

What we promise...

We will show, how to write a sample application, that makes use of the new features of PHP-GTK2

Let's start building

We will begin our work with writing our main class `Application`, which we present on the Listing 1 and 2. At first, it will create the main window, instantiating the `GtkWindow` class as the window object, then set its basic parameters (size, position and title). Next, we will set the container, that organizes the positioning of our widgets. For this purpose, we will use the `GtkVBox` class, which we'll instantiate as `$vbox`.

It's time for the pulldown menus we've mentioned before. Here we are with the `GtkMenuBar` class; we'll create its object named `$MenuBar`. Next, using the `GtkMenuItem` class, we'll create the menu bar positions called *File*, *Edit* and *Help*. They will be the separate objects, which we'll not add to the window yet.

The Main Menu

The next thing we'll do is to create the respective menus, that open when we click at one of the mentioned positions. For this, we have the GtkMenu class, which we'll instantiate as *$SubMenuFile*. But wait, we want the positions in those menus to have icons! Fortunately, we don't have to create them by ourselves: PHP-GTK2 gives us the Stock Images, that is: the set of the essential, commonly used icons for our buttons, menus, listings and others, depicting the operations like save, open, close, delete, add, clear, quit, yes, no, etc. They have a clear naming convention, so we don't need to worry in each case about what icon to use. Each item belonging to the `GtkMenu` will be an object of the `GtkImageMenuItem` class, with an icon passed as an argument to its constructor. If you miss the icon name, the position will simply have no picture. In the menu *File* we add the positions called *Clear*, *Add*, *Edit* and *Quit*. Clicking on the first one will start the database, drop the products table and create it again. The second one, *Add*, will show the user a dialog containing a form allowing to add a product, while the third one, *Edit*, will show him the list of products, that allows to edit a chosen position. Finally, clicking on the position Quit will cause application to stop its operation.

In the next menu, called *File*, we will have only one option: *Edit*. It will start the embedded text editor. Similiarly, there will be one and only option in the Help menu,

Figure 1. *Menus of our application*

and it will be called *Help*. Clicking on it will pop up the well-known *About* dialog window.

All menus are shown on the Figure 1.

The next thing is to connect the signals send by the menu items on the event of being clicked with the callback methods, that will perform the revelant operations. Those methods will be known as `onClear()`, `onAdd()`, `onList()`, `onEdit()` and `onHelp()`. We do this using the `connect()` method of each item. Then we'll add the items to their respective menus. The last thing to do about the menus is to append the created menus to the Main Menu. We'll do this in two steps: first, we will append the submenus for each menu bar position using the `set_submenu()` method of each of the latter, and second, we'll append the three positions to the Main Menu Bar (`$MenuBar`).

The last things left in the constructor are: to add our `$vbox` container to the window, add the widgets to it, and show all elements of the window with the `$window->show_all()` method.

The Callback Methods and the Message Dialogs

Now, it's a high time to start creating the callback methods for the respective menu items, that we've mentioned before. As the first, we will create `onClear()`.

In the beginning, it will check whether the database we want to clear exists. Then, it should ask us, if we really want to remove all stored records. For this, PHP-GTK2 offers us a flexible Dialog Box (`GtkMessageDialog`), that can be used in many situations (warning messages, error messages, confirmation dialogs, input boxes and so on). We will use it into the `MODAL` mode, what means, that it persists on the screen and prevents us from performing other operations within our application till it's closed. We'd like to have a question mark on it: no problem, PHP-GTK2 gives us the ability to use the very stock icons we used before, and some sets of the stock buttons as well. For the latter, we'll need the *YES/NO* buttons. See the Figure 2. Then we'll wait for the user's response: should it mean allowing us to remove the database, we'll make a connection to our SQLite base, drop the products table and, finally, create it again. After the database is properly cleared, the user will see another dialog, shown on the Figure 3, that

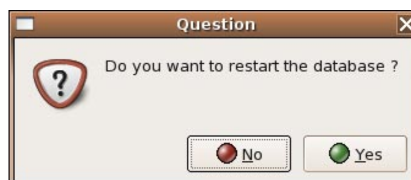


Figure 2. *A dialog letting us to decide, whether we want to restart a database*

Figure 3. *Information, that the database has been cleared*

informs about the success.

Listing 1. *The Main Interface: the main window of our application, stored in the file product.php*

```
<?php
// Class Application - encapsulates the main interface
class Application{
   private $window;
   function __construct(){
      // creates the main window and sets its parameters
      $this->window = new GtkWindow;
...
      $vbox = new GtkVBox;
      // creates the menubar
      $MenuBar = new GtkMenuBar;
      // menu options
      $MenuFile = new GtkMenuItem('_File');
...
      // sub-menu File with stock-itens
      $SubMenuFile = new GtkMenu;
      $ItemFile1= new GtkImageMenuItem(GTK::STOCK_CLEAR);
...
      $ItemFile4= new GtkMenuItem;
      $ItemFile5= new GtkImageMenuItem(GTK::STOCK_QUIT);
      // connect menu options to the methods
      $ItemFile1->connect('activate', array($this, 'onClear'));
...
      $ItemFile5->connect('activate', array($this, 'onQuit'));
      // append the items to the sub-menu
      $SubMenuFile->append($ItemFile1);
...
      // sub-menu edit
      $SubMenuEdit= new GtkMenu;
      $ItemEdit1= new GtkImageMenuItem(GTK::STOCK_EDIT);
      $ItemEdit1->connect('activate', array($this, 'onEdit'));
      $SubMenuEdit->append($ItemEdit1);
      // sub-menu help
...
      $MenuFile->set_submenu($SubMenuFile);
...
      $MenuEdit->set_submenu($SubMenuEdit);
...
      $this->window->add($vbox);
      $vbox->pack_start($MenuBar, false, false);
      $this->window->show_all();
   }
   // Method onClear - creates the database structure
   function onClear(){
      if (file_exists('data.db')){
         $dialog = new GtkMessageDialog(null, Gtk::DIALOG_MODAL,
            Gtk::MESSAGE_QUESTION, Gtk::BUTTONS_YES_NO,'Do you want to restart
            the database ?');
         $response = $dialog->run();
         $dialog->destroy();
         if ($response == Gtk::RESPONSE_YES){
            // drop the table products
            $conn = sqlite_open('data.db');
...
         }
         else if ($response == Gtk::RESPONSE_NO){
            return;
         }
      }
      // create the table products
      $conn = sqlite_open('data.db');
      $sql = 'CREATE TABLE products (code,description,' .'unit,amount,cost,pric
                 e)';
      sqlite_query($conn, $sql);
```

Adding New Products: The Products Form

The next callback method is `onAdd()`, which allows us to add the new item. It invokes the dialog shown on the Figure 4. In fact, all this method does is to make use of a class named ProductNew, that is stored in the file *ProductNew.class.php* (see Listing 3). It would be too lengthy and boring to describe it in the full detail, so we'll focus on the most important issues related to it.

First, in its constructor it creates the same object of the `GtkWindow` class as we-'ve seen before. It uses labels (`GtkLabel`) for showing descriptions. The most interesting thing about it is, that we can format their contents. In PHP-GTK2 it's much easier and more flexible than in PHP-GTK1, when it took much effort to make the text look like we wanted to (even for the simple effects as Bold, Italic, Underline or colors). PHP-GTK2 uses Pango, a GTK open-source framework that deals with all layout and rendering stuffs. With Pango it's possible to use a markup language derived from SGML to format text for exhibition, a simple way to define style and colors (see Figure 5). Despite it's not shown here, it's possible to write text vertically or using angles (expressed in degrees). In our dialog, we will create the following labels: *Code*, *Description*, *Amount*, *Unit*, *Cost* and *Price*.

The bottommost label will contain a hint for the user, starting that the fields *Code*, *Description* and *Price* must be filled. We would like, however, that it wouldn't be visible all the time, as more experienced

History

In its first generation, PHP-GTK was a language binding beween PHP-4.x and Gtk-1.2.

This version, that is still used by the majority of projects, was hindered by the deficiencies of the PHP4's object model. As soon as PHP5 was launched, The PHP-GTK creator's Andrei Zmievski, started a hard task to rewrite the code base of PHP-GTK to use the new facilities of PHP5 and Gtk2. The main part of the work was to take advantage of the new PHP's object model and Gtk2's new features, that involves the new flexible type system (as PHP) and a good extensibility, what makes easier the task of mapping Gtk2 API onto PHP's object model. It's important highlight that PHP-GTK2 is a work in progress that is on an alpha stage.

Figure 4. *A form for adding a new product*

users might not need it. With PHP-GTK2 it's no problem: we have the component called `GtkExpander`, which is a container allowing to expand or collapse its content. We will use it for our help label. Let's declare an instance of `GtkExpander` class first (we'll call it `$expander`), then create a label named `$help`, and add the label to `$expander`. By default, it will be expanded (using the method `set_expanded(true)`).

Another widget we'll use is text entry (`GtkEntry`) for inputting the text. We will have a text entry for each label. Here, PHP-GTK2 offers us a useful feature called auto-completion. It allows to link a `GtkEntry` widget with a `GtkListStore`, a data model with possible values to be used for auto-completion while the user types, using an object of the third class called `GtkEntryCompletion`. It's very common in Gtk applications, specially for File Dialogs to suggest the files that matches with the typed value (see Figure 6). In our dialog, we'll use EntryCompletion in the Unit field to list a set of common used units. For this purpose, apart from creating the object for the `GtkEntry` (`$this->entries[3]`), we will instantiate the `GtkListStore` class as `$store`. Next, we will add the common units using the `append()` method of the `$store` object. Then we'll create the object `$completion` of the GtkEntryCompletion class , set the `$store` object as its source of auto-completion data and, finally, apply the completion feature to the Unit entry, using the `set_completion()` method of the entry.

The last thing we need is the Save button, which will allow us to add the new item to the database. For this purpose, we will instantiate the class `GtkButton` (as `$button1`). On the button, we will place a stock icon depicting a diskette. Then, we'll connect the clicked event of the button with the callback method named `onSaveClick()`. The latter will read the values inputted in the form, store them into the database and clear the form through another method we create, named `Clear()`.

Figure 5. *Text effects in PHP-GTK2*

Listing 2. *The Main Interface: the main window of our application, continued*

```
      // show the success message to the user
      $dialog = new GtkMessageDialog(null, Gtk::DIALOG_MODAL, Gtk::MESSAGE_INFO,
         Gtk::BUTTONS_OK, 'Database clear');
      $response = $dialog->run();
      $dialog->destroy();
   }
   // Method onAdd - creates the Products Form
   function onAdd(){
      include_once 'ProductNew.class.php';
      new ProductNew;
   }
   // Method onList- show the Products Listing allowing the user to change the
                      data
   function onList(){
   include_once 'ProductList.class.php';
   $obj = new ProductList;
   $obj->Show();
   $obj->showData();
   }
   // Method onEdit - Opens a little Text Editor
   function onEdit(){
      include_once 'TextEditor.class.php';
      new TextEditor;
   }
   // Method onHelp - Show the About Dialog
   function onHelp(){
      include_once 'AboutDialog.class.php';
      new AboutDialog('Products', 'This software is open-source');
   }
   // Method onQuit - Quit the Application
   function onQuit(){
      Gtk::main_quit();
   }
}
// creates a new instance of Application
new Application;
Gtk::Main();
?>
```

Figure 6. *Auto-completion*

Show me the products or why we like the trees

The next callback method in our Application class is `onList()`, which is being called when the *Edit* position in the *File* menu has been clicked. It makes use of another class, `ProductList`, stored in the file *ProductList.class.php*. See Listing 4 and 5. There is no need to describe it in detail, so, as in the cases of previous method, we will focus on its most important components instead. The main task of the class is to show the editable list of all items in our store. On edition, it will update the data directly to the database. For this purpose, we will use the brand new `GtkTreeView` component that PHP-GTK2 provides us with, and which is the huge step forward compared to the previously existing, simple components as `GtkCTree` and `GtkCList`. The latter widgets allowed us to display only the trees (the first one, see Figure 7) or lists (the second one, Figure 8), `GtkTreeView` permits us to display both types, in the meanwhile grossly extending their functionality. For example, now we can add check boxes and images to the items. What is interesting to us, both tree and list can have columns, and the data it contains may be edited, as it was presented on a spreadsheet.

We will use `GtkTreeView` in the mode of an editable list with columns. First, we'll instantiate it as `$this->list`. Next, we'll provide it with the data. It's important to know, that the `GtkTreeView` allows the total separation of Model, View and Controller layers in the MVC (*Model-View-Controller*) pattern. That means, that the data model is completely separated from its screen exhibition. The data are stored in the *Model*, that can be a `GtkListStore` (for listings) or `GtkTreeStore` (for trees). We'll use `GtkListStore`, instantiating it as `$this->model`. Next, we'll create the columns for *View*, to display the data. For this purpose, we have the separate `GtkTreeViewColumn` class. For each of them we'll set the title, and create and add (using the `connect()` method of each renderer) a renderer, which is responsible for visualisation of the data stored in the

Listing 4. *The code of the ProductList Form, stored in the file ProductNew.class.php*

```
<?php
// Class ProductNew - Product Form for inclusion
class ProductNew extends GtkWindow{
...

   // Constructor Method - Creates the window and all the entries
   public function __construct(){
...
      // creates all the labels and entries, line by line
      $this->labels[0]   = new GtkLabel('<span foreground="red"><b>Code
         </b></span>');
      $this->entries[0]  = new GtkEntry;
      $this->entries[0]->set_size_request(80,-1);
...
      // Creates the DataModel
      $store = new GtkListStore(Gtk::TYPE_STRING);
      // Append values to the DataModel
      $store->append(array('UN'));
...
      // Creates the EntryCompletion
      $completion = new GtkEntryCompletion();
      $completion->set_model($store);
      $completion->set_text_column(0);
      $this->entries[3]->set_completion($completion);
...
      // pack all the labels and entries to the vertical box
...
      // create the save button box
      $save_box= new GtkHBox();
      $button1  = GtkButton::new_from_stock(Gtk::STOCK_SAVE);
      $button1->connect('clicked', array($this, 'onSaveClick'));
      $save_box->pack_start(new GtkHBox, true);
...
      $expander = new GtkExpander('<b><i>Help</i></b>');
      $expander->set_use_markup(true);
      // Put a little help text inside the expander
      $help = new GtkLabel;
      $help->set_alignment(0.2, 0.5);
      $help->set_markup('<b><u>Filling the form</u></b>In order to fill the
         form you must not leave Code, Description or Price empty...');
      $expander->add($help);
      $expander->set_expanded(true);
      $vbox->pack_start($expander, false, false);
      parent::add($vbox);
      parent::show_all();
   }

   // Method onSaveClick -  Save the screen data to the database
   public function onSaveClick(){
   // reading data from entries and writing it to the DB
      $conn = sqlite_open('data.db');
      $product->code           = $this->entries[0]->get_text();
      $sql = "INSERT INTO products (code, description, amount," .
...
...
   }

   // Method Clear - Clear all the entries from the form.
   private function Clear(){
      for ($n=0; $n<=5; $n++){
         $this->entries[$n]->set_text('');
      }
      // focus the cursor to the first field
      parent::set_focus($this->entries[0]);
   }
```

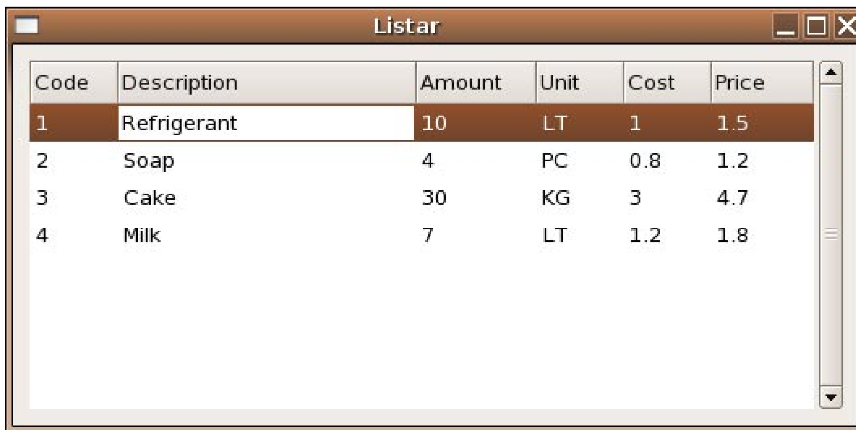


Figure 7. *GtkTreeView in the tree mode*

Figure 8. *GtkTreeView in the list mode*

Listing 4. *The code of the ProductList, which is stored in the ProductList.class.php file*

```
<?php
// Class ProductList - Product Listing
class ProductList extends GtkWindow
{
   private $window;
   private $model;
   private $list;
   public function __construct(){
...
      // creates the treeview
      $this->list = new GtkTreeView;
      $scroll->add($this->list);
      // creates the model, with 6 elements
      $this->model = new GtkListStore(Gtk::TYPE_STRING, Gtk::TYPE_STRING,
         Gtk::TYPE_STRING,Gtk::TYPE_STRING, Gtk::TYPE_STRING, Gtk::TYPE_STRING);
      // creates the columns
      $column1 = new GtkTreeViewColumn();
...
      $column1->set_title('Code');
...
      // defines the renderers
      $cell_renderer1 = new GtkCellRendererText();
...
      // connects the renderers when the user edit the data
      $cell_renderer2->connect("edited",array($this,'onEdit'),1,'description');
...
```

model. For the last one, we will use the `GtkCellRendererText` class, which allows for displaying the text; there are some other renderers to show other datatypes, as image and checkboxes, and each column can pack one or more of them. When the renderers are added, we will configure them properly, setting, among others, the width and the editability of the column, and connecting the callback function `onEdit()` with an event edited, which happens on edition of any of the elements.

The next step will be to add the columns to our list, what we'll do using the `append_column()` method of our list (`$this->list` object).

Now it's important to add the *Model* to the list, which we'll achieve with the `set_model()` method of `$this->list` object. It contains no data yet, because it will be added later. Besides, a huge benefit of this separation of layers in `GtkTreeView` is, that one data model can be utilized by one or more `GtkTreeView` objects, what allows us to have the same data displayed in various parts of our application. Our `ProductList` class is almost ready. What we need yet, is the method that will fill our model with data. We'll name it `showData()`, and it will be called from inside our `Application` class. At first, it will make a database connection and fetch the data. Then, it will include the data into the model, using the `set()` method of `$this->model`.

Another necessary method is the callback `onEdit()`. It will be called, when the user edits any cell. It will perform an automatic update to the database, recognizing the correct row in the DB table by the product code. And that's all about the `ProductList` class.

The Editor

Another thing we want to add to our project is the simple text editor, which opens after the Edit option of the Edit menu is clicked, and the `onEdit()` method of the Application class is being called. Beside edition of text, it should allow us to load and save it. It will look as on the Figure 9.

We will create a class `TextEditor`, which will be stored in the file *TextEditor.class.php*. We present it on the Listing 6. Its main component will be the text window. For this purpose, we will use the classes `GtkTextView` and `GtkTextBuffer`. In PHP-GTK1 we used GtkText component that is deprecated now. An important thing is, that unlike PHP-GTK1, which

Figure 9. *A simple text editor*

Listing 5. *The code of the ProductList, continued*

```
      // pack the renderers
      $column1->pack_start($cell_renderer1, true);
...
      // define the width
      $cell_renderer1->set_property('width', 50);
...
      // allow the user to edit the data
      $cell_renderer2->set_property('editable', True);
...
      // define the model position that the renderers will be linked
      $column1->set_attributes($cell_renderer1, 'text', 0);
...
      $this->list->append_column($column1);
...
      $this->list->set_model($this->model);
      $this->list->show_all();
...
   }
   // Method showData - List all Products from database
   public function showData(){
      // opens the database and reads all data in the while() loop
      $conn = sqlite_open('data.db');
      $query = sqlite_query($conn, 'select code, description, amount,'.
         ' unit, cost, price from products');
      while ($data = sqlite_fetch_array($query)){
         $iter = $this->model->append();
         $this->model->set($iter, 0, $data['code'],
            1, $data['description'], 2, $data['amount'], 3, $data['unit'],
            4, $data['cost'],        5, $data['price']);
      }
      sqlite_close($conn);
   }
   // Method onEdit - Called when the end user changes the data
   public function onEdit($cell_renderer, $path, $new_text, $column_number,
      $column_name){
      // get the selection
      $treeselection = $this->list->get_selection();
      list($model, $iter) = $treeselection->get_selected();
      // set the new value to the iterator
      $model->set($iter, $column_number, $new_text);
      // get the first column
      $code = $this->model->get_value($iter, 0);
      // opens the database and executes the update
      $conn = sqlite_open('data.db');
      $query = sqlite_query($conn, "update products set " .
         "$column_name='$new_text' where code='$code'");
      sqlite_close($conn);
   }
}
?>
```

used to merge the content and its visualisation in the same object, in PHP-GTK2 we store the data outside the widget, in the `GtkTextBuffer` class we mentioned about. So, the same text can be shown by different `GtkTextView` widgets, the same way it happens with `GtkTreeView` widget. `GtkTextBuffer` also offers us iterators, that is the possibility to bookmark some positions in the text.

We'll instantiate the `GtkTextView` as `$this->textview` in the constructor of the class. We will do the same to the `GtkTextBufffer`, creating an object `$this->textbuffer`. Next, we will join the data model with the widget, using the `set_buffer()` method of `$this->textview`.

As we said, we want to be able to save and load (open) the text files. As we seen it before on the Figure 9, the editor will have a toolbar containing two positions (with the proper icons): *save* and *open*. To create the toolbar, we'll use the `GtkToolbar` class, instantiating it as `$toolbar`. Next, we will make the buttons for saving and loading the data, creating the respective objects of `GtkToolButton` class. We'll set the label and stock icon for each. The next step will be to connect the event of clicking each button (`clicked`) with the respective method. For loading it will be `openFile()`, while for saving we'll have a method called `saveFile()`. The main part of our editor is ready: let's take to the event methods.

In the first of them, `openFile()`, we want to be able to choose a file from the list. For this purpose, PHP-GTK2 gives us `GtkFileChooserDialog`, which replaces the `GtkFileSelection` from PHP-GTK1, being more useful and easier to extend (see Figure 10). It binds to `GtkFileChooser` from the Gtk2 library, which is being used practically by all Gnome-based applications, such as Evolution, Gnumeric, Gaim, Gimp and others. We will instantiate `GtkFileChooserDialog` as `$dialog`, giving it a caption *Opening the file*, and attaching a set of responses, consisting of *OK* and *CANCEL*. For each of the responses there is a stock icon, too. If we confirm our will to open the file, the text buffer will be cleared (with `$this->textbuffer->delete()`) and the contents of the file will be inserted at the cursor (`$this->textbuffer->insertatcursor()`). After that, the dialog will be destroyed.

Another method is called `saveFile()`. Here, we will utilize the same

Listing 6. *The code of the text editor, stored in the file TextEditor.class.php*

```
<?php
// Class Editor - A little text exitor
final class TextEditor extends GtkWindow{
   private $textview;
   private $textbuffer;
   function __construct(){
      // creates the window
...
      // Creating the Toolbar
      $toolbar = new GtkToolbar;
      // Creating the save and open buttons
      $save = new GtkToolButton;
      $save->set_label('save');
      $save->set_stock_id('gtk-save');
      $save->connect('clicked', array(&$this, 'saveFile'));
...
      // Insert the buttons on toolbar
      $toolbar->insert($open, 0);
      $toolbar->insert($save, 0);
      $vbox->pack_start($toolbar, false, false);
...
      // Creating the duo TextView/TextBuffer;
      $this->textview = new GtkTextView;
      $this->textbuffer = new GtkTextBuffer;
      $this->textview->set_buffer($this->textbuffer);
...
   }
   // Method openFile - Shows the FileDialog and puts the file contents
   // inside the TextBuffer
   public function openFile(){
      // Creates the FileChooserDialog
      $dialog = new GtkFileChooserDialog('Opening the file', NULL,
         Gtk::FILE_CHOOSER_ACTION_OPEN,array(Gtk::STOCK_OK,Gtk::RESPONSE_OK,
         Gtk::STOCK_CANCEL, Gtk::RESPONSE_CANCEL));
      // Shows the FileChooserDialog
      $response = $dialog->run();
      if ($response == Gtk::RESPONSE_OK){ // if the user has clicked OK
         //Clear the TextBuffer and Insert the file's content into the TextBuffer
         $first = $this->textbuffer->get_start_iter();
         $end   = $this->textbuffer->get_end_iter();
         $this->textbuffer->delete($first, $end);
         $this->textbuffer->insert_at_cursor(file_get_contents(
            $dialog->get_filename()));
      }
      $dialog->destroy();
   }
   // Method saveFile - Shows the FileDialog and write the TextBuffer
   // contents to the selected file.
   public function saveFile(){
      // Creates the FileChooserDialog
      $dialog = new GtkFileChooserDialog('Saving the file', NULL,
         Gtk::FILE_CHOOSER_ACTION_SAVE,array(Gtk::STOCK_OK,Gtk::RESPONSE_OK,
         Gtk::STOCK_CANCEL, Gtk::RESPONSE_CANCEL));
      // Shows the FileChooserDialog
      $response = $dialog->run();
      if ($response == Gtk::RESPONSE_OK){ // if the user has clicked OK
         // Get the TextBuffer content and write it to the file
         $first = $this->textbuffer->get_start_iter();
         $end   = $this->textbuffer->get_end_iter();
         $text  = $this->textbuffer->get_text($first, $end);
         file_put_contents($dialog->get_filename(), $text);
      }
      $dialog->destroy();
   }
}
?>
```

`GtkFileChooserDialog` for chosing the name and location of the file to save. As in the previous case, it will have two buttons: *OK* and *CANCEL*. When the user clicks on *OK*, the method will read the text from the buffer using `$this->textbuffer->get_text()` method. Next, it will save the file at the chosen (or inputted) name, using `file_put_contents()`. After that, as in the previous method, the dialog will be destroyed.

Who Made Me: the Use of About Dialog

The Last thing to do in our interface is to create the About dialog, that will pop up after the Help position in the Help menu is clicked. We will create a class named `AboutDialog` and store it in the file *AboutDialog.class.php*. We present it on the Listing 7.

Here we will make the use of exceptions and image manipulation. As usually, the constructor will start with setting some parameters for the dialog. Next, we want to load an image named gnome.png and display it in the dialog (see Figure 11). For this purpose, we will use teh `GdkPixbuf` class. It deals with the most popular image formats like PNG, JPEG and others, what releases us from the task of additional image converting (eg. with the use of third-party tools). In the PHP-GTK1, this class existed, but its use was optional. It's not optional anymore in GTK2, having become a part of the set of available native classes. It can be used for all image manipulation tasks, including the use as icons in the toolbars and menus. And we don't need to employ the XPM (XpixMap) format anymore, which was the case with PHP-GTK1, if `Gdkpixbuf` wasn't used.

Thus, we will instantiate the GdkPixbuf class as $pixbuf, passing the path of the file *gnome.png* to its constructor. Next, we will create an image widget of the class `GtkImage`. We will name it as `$imagem`. Using its method `set_from_pixbuf()`, we will copy the content of `$pixbuf` to it.

We want to be sure, however, that the file will load, and if it doesn't exist, that it will not cause application to terminate by error. Therefore, we'll embrace all of those bitmap-related operations in the `try{}` block of the `try{}..catch{}` construction. PHP-GTK2 cooperates with the exception system, which has been introduced in PHP5, generating exceptions under some situations like errors occuring during the

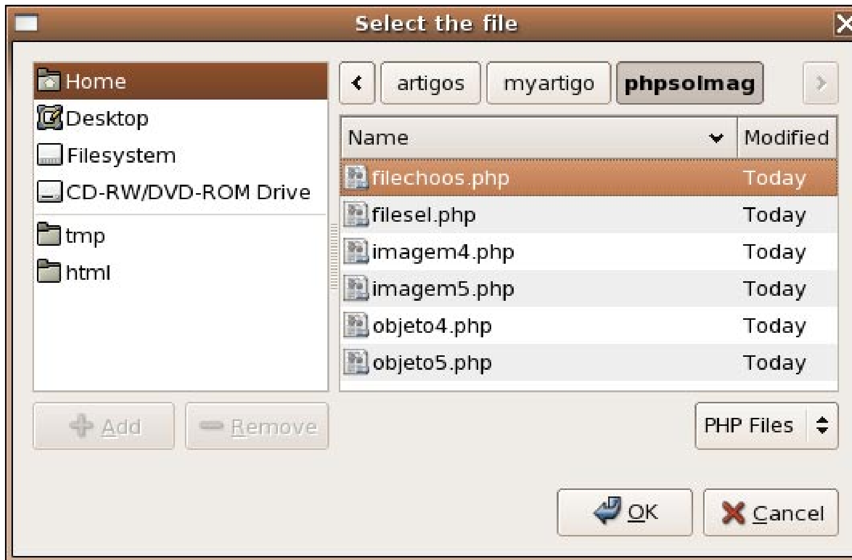


Figure 10. *File selection*

Listing 7. *The code of the AboutDialog class, stored in the file AboutDialog.class.php*

```
<?php
// Class AboutDialog - Show the Information about the current application
final class AboutDialog extends GtkWindow{
   public function __construct($software, $text){
...

       // try to load the image
       try{
          $pixbuf = GdkPixbuf::new_from_file('images/gnome.png');
          $imagem = new GtkImage;
          $imagem->set_from_pixbuf($pixbuf);
          $vbox->pack_start($imagem);
       }

       catch( (PhpGtkGErrorException $error)
          // if any error ocurrs, show it on the screen
          $dialog = new GtkMessageDialog(null, Gtk::DIALOG_MODAL,
          Gtk::MESSAGE_ERROR,Gtk::BUTTONS_OK, $error->message);
          $response = $dialog->run();
          $dialog->destroy();
          return;
       }

       // show the text about the application
       $this->label = new GtkLabel($text);
       $vbox->pack_start($this->label);

       // creates the "Close" button
       $this->button = new GtkButton('Close');
       $this->button->connect('clicked', array($this, 'onClose'));
       $vbox->pack_start($this->button, false, false);
       parent::show_all();
    }

    // Method onClose - Closes the window
    public function onClose(){
       parent::destroy();
    }
...
}
?>
```

Figure 11. *The About dialog*

object construction, in methods that use the Gerror mechanism, such as static constructors like `GdkPixbuff::new_from_file()` and in codepage conversions. In our case, if Logo is not found, it'll generate a exception controlled bye the main flow of our AboutDialog class.

Our application is ready and you may test it.

Summary

The example we shown is quite simple, but it illustrates the most important novel aspects of PHP-GTK2; the best way to learn more is to start expanding it. PHP-GTK2 is a huge step forward, compared to the PHP-GTK1, and PHP5 plays an important part in making it a successful interface: without its new, better object model many things would simply be impossible. Together, PHP-GTK2 may make PHP a serious choice for the programmers, who make the client-side, GUI applications.

What you should know...

Pablo Dall'Oglio (pablo@dalloglio.net) is author of the first book about PHP-GTK of the world. He is also author of Agata Report (*http://www.agata.org.br*) Tulip Editor (*http://tulip.solis.coop.br*) and Coordinator of GNUTeca project (*http://www.gnuteca.org.br*) (an open source software for library management).