

This article was originally written for the February 2005 edition of International PHP Magazine.

Free Your CMS

David Heath

Use PHP-GTK to free your CMS users from the need to be on-line

David Heath

Web-based CMS systems are commonplace. But in many cases a Web-based CMS system does not effectively serve the needs of the end user. If they are travelling or have poor Internet connectivity, then working with an on-line CMS can be awkward and frustrating. In this article *David Heath* explains how he freed his users from the need to be on-line. He built an off-line desktop client tool to complement his organisation's Web-based CMS system. The end product is a synthesis of several exciting technologies: PHP-GTK, SQLite, and XML-RPC

Introduction

OneWorld.net is a multi-lingual Web portal with editorial offices around the world, including offices in Lusaka, Zambia and New Delhi, India. Use of our Web-based CMS hosted in London, UK is often made difficult by poor quality Internet connectivity in those regions. To overcome these problems, we built an "off-line working tool" (OWT) for our CMS using PHP-GTK.

In this article I will cover:

1. goals of the project
2. how the desktop client and user interface were built
3. the off-line data store
4. design of the replication and remote communication
5. evaluation and conclusions

Goals of the project

The main goal of the project was to give our journalists a tool with which they could write their articles while off-line. The tool should store the articles locally and allow them to be sent in later when connectivity becomes available. Since this was a pilot project, we kept the scope of the project very focused on satisfying this one need as fully as possible, avoiding any additional functionality.

In satisfying this design goal, we were still left with a challenging set of requirements for our tool:

- the tool needs to keep up to date with changes in the central Web-based CMS (e.g. changes to folder organisation)
- it must be optimised for a low-bandwidth Internet connection. In practice, this means that any data updates should be incremental.
- it must gracefully withstand network drop-outs, data corruption, etc.

As well as these functional requirements, we also wanted the tool to be cross-platform (Windows and Linux), lightweight, and easy to install.

Options for implementation

Before we get stuck into the technical details, I'll run through a few of the other technologies we considered using. There are clearly many viable ways to tackle this problem, and each approach has its own strengths and weaknesses. I don't claim that the approach we took is necessarily the "best" for all circumstances, but it fits well with our existing software and the skills of our team.

The first option we considered was Mozilla/XUL/Javascript as the basis of our cross-platform GUI tool. This clearly has a lot of mileage and is a serious contender for these kind of applications. We rejected it for a couple of reasons:

1. We are not javascript experts, nor are we familiar with the Mozilla architecture. There seemed to be a great deal to learn and quite a high barrier to entry. The risk was too high.

2. I couldn't identify any mechanism I could easily use for local storage. While I am sure there are options, none reached out and grabbed me (since that time there seems to have been some movement on creating an XPCOM wrapper for SQLite).

Another option we considered was to put a lightweight web server such as Nanoweb on the client and develop using a regular web app paradigm. This option was rejected for having too many "moving parts" and therefore a potential for maintenance problems.

The final option we considered was to use Java, Kylix, or something similar. However, we wanted to build something using 100% open source components, using our favourite language!

The PHP solution

PHP-GTK seemed like a good option for us, satisfying all of our criteria outlined above. In particular:

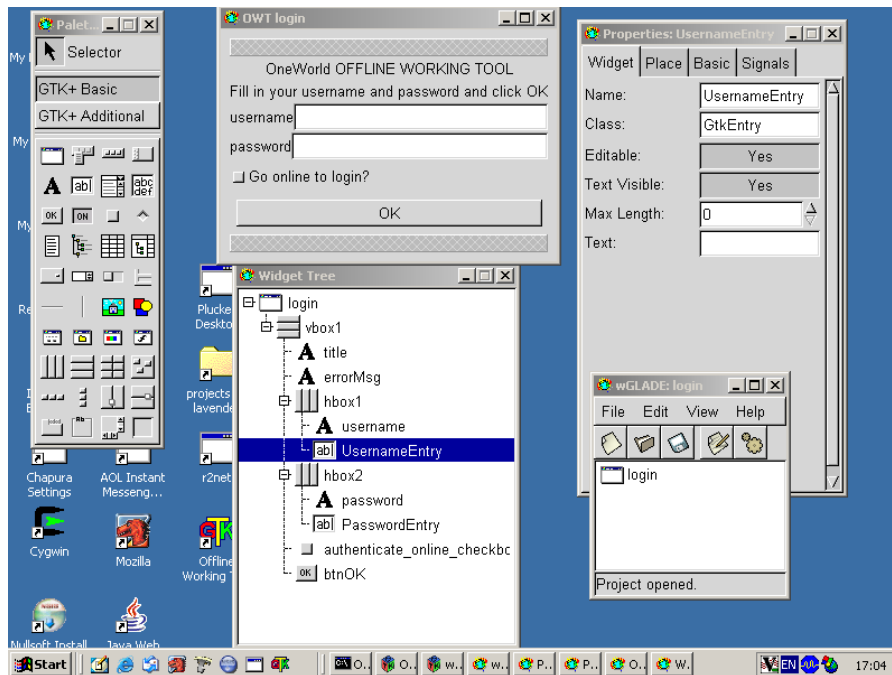
- We can re-use code from our on-line CMS
- Keeps focus on our preferred technologies, less training needed
- Light weight (runtime + code is 4.5MB)
- Fully open source
- More fun!

The only potential downside of PHP-GTK was that it may be less "stable." Although it has been used for some fairly sophisticated applications (the phpmole IDE written by Alan Knowles is perhaps the best example), it hasn't really been used in any widely deployed, mainstream app. This means it has not had as much real-world testing. Having said that, both PHP and GTK have separately been widely used, which goes quite a long way to increase confidence. I'm happy to say that we haven't experienced stability problems to date.

Rapid development using Glade

We chose to use Glade as a rapid application development (RAD) tool for developing the user interface of our tool. Glade is a graphical tool for designing GTK windows and components, it's something like Visual Basic for GTK. Glade stores descriptions of the user interface in XML files, which are read in at runtime using the *libglade* library. Libglade will generate a window layout for you based on the XML files, saving quite a lot of fiddly coding to build your window layout.

One thing you should note is that PHP-GTK uses Glade version 1, and files using Glade version 2 will not work with PHP-GTK. Most Linux distributions ship with packages for both versions (normally called "glade", and "glade2"). Under Windows you should download wGlade from <http://wingtk.sourceforge.net/>.



Screenshot 1: Glade running under MS Windows. Clockwise from left: palette containing widgets to be added to your window, login window being edited, properties sheet for a widget, main glade window, widget tree showing hierarchy of components.

Glade is an excellent tool to experiment with in order to understand how GTK windows are put together and what each of the different types of widgets can do. Why not give it a try now?

Your first Glade application

So, hopefully you've managed to fire up Glade and create a sample window. Making use of it in your PHP application is equally straightforward. Listing 1 gives a basic example (the corresponding Glade file can be found in the *glade_example* subdirectory of the sample code).

On line 10, the Glade file is loaded. At this point, libglade will read and parse the *.glade* XML file. On line 13, we call *signal_autoconnect()* on our *GladeXml* object to ask libglade to connect all of the event callback handlers. The handlers must be set up by the user in Glade itself – this is done on the “signals” tab in the property window (see Screenshot 2). When called with no arguments, *signal_autoconnect()* will treat the handler name as the name of a PHP function and connect the signal to the corresponding PHP function.

In our case, we will follow an object-oriented approach, so this basic form is not suitable. Instead, we need to pass a “handler mapping” to *signal_autoconnect*. We are not restricted to giving just a function name as a callback, but can use any “callable thing,” including the *array(\$objectInstance, \$methodName)* syntax. See the PHP documentation for the PHP built-in function *call_user_func()* and the *callback pseudo* type for an explanation.

Listing 2 shows a semi-automatic way of connecting up handlers for methods on the current object instance. Note the slightly unexpected structure of this handler map: *array(string handlername => array(callback \$callable), ...)*. It seems slightly unnecessary to map onto an array containing a single callback .. why not just map directly onto the callback? In fact, the array is used so that you can pass additional parameters to your callback if desired. To do this you would pass a structure of the form *array(string handlername => array(callback \$callable, \$param1, \$param2, ...), ...)*, *\$param1* would be passed as the first parameter to your callback.

So, we now have the ease of use of Glade at our finger tips. One area which I won't dive into further is that of internationalisation for your Glade app. If you want to read about this there is an up-to-date tutorial at: <http://gtk.php.net/manual/en/tutorials.translation.glade.php>

Listing 1: A basic Glade example

```

<?php
if (!extension_loaded('gtk')) {
    $ok = @dl('php_gtk.'.PHP_SHLIB_SUFFIX);
    if (!$ok)
        die('unable to load php gtk');
}

// Load the glade file
$glade =& new GladeXml("glade_example.glade");

// Automatically connect signals
$glade->signal_autoconnect();

// Get the widget corresponding to the main
// window and show it
$window =& $glade->get_widget('window1');
$window->show_all();

// Run the GTK main loop
Gtk::main();

/**
 * Callback handlers
 */
function on_italic_activate() {
    print "italic";
}

function on_bold_activate() {
    print "bold";
}

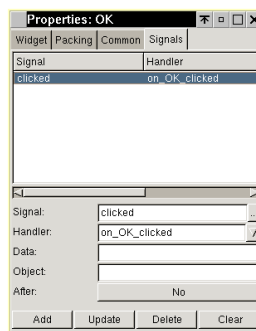
function on_OK_clicked() {
    print 'OK';
}

function on_window1_destroy() {
    Gtk::main_quit();
}
?>

```

END LISTING 1

Screenshot 2: Editing signal handlers



Listing 2: Glade signal connection, OO-style

```

class TestGladeWindow {
    function show() {
        $glade =& new GladeXml(
            "glade_example.glade");

        $handlers =
            array('on_italic_activate',
                'on_bold_activate',
                'on_OK_clicked');

        $handlerMap = array();
        foreach ($handlers as $handler) {
            $handlerMap[$handler] =

```

```

        array(array(&$this, $handler));
    }
    $glade->signal_autoconnect_object(
        $handlerMap);
    $window =& $glade->get_widget('window1');
    $window->show_all();
}

/* ...handlers here... */
}

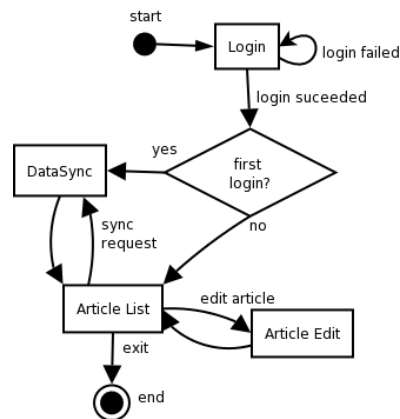
```

End listing 2

GUI controller

The off-line working tool (OWT) consists of multiple dialogs and windows which might appear in different orders at different times. The flow of execution is shown in Figure 1.

Figure 1: Flow of execution in the off-line editor



Each rectangular box in Figure 1 corresponds to a window in our GTK app, which also has a corresponding PHP class. In the OWT sources you can find these classes in *lib/ezowt/gui*. Each window class is responsible for displaying itself, populating relevant widgets and handling the user interaction for that window. We also need some way of managing the flow of control between different windows. For a good design, we should avoid spreading that responsibility between the window classes.

Instead, we follow the Model-View-Controller (MVC) pattern and create a controller class which takes responsibility for the transitions between windows. The controller follows the singleton pattern (i.e. there is exactly one controller in the whole system which is accessed via *eZowtClientController::getGlobalController()*). When the program starts, we call *\$controller->start()* to commence the main flow of application execution. Each state transition in Figure 1 corresponds to a method call on the controller. On each transition, the controller will open and close windows as needed according to the logical flow of the application.

Listing 3: Controller

```

class eZowtClientController {
    /* Access the one and only
     * controller instance
     */
    function &getGlobalController() {
        static $controller;

        if (!isset($controller)) {
            $controller = new eZowtClientController();
        }
    }
}

```

```

    return $controller;
}

/* the 'start' state transition message */
function start() {
    // perform post installation checks
    $ok = eZOWtPostInstall::check();
    if (!$ok) {
        print "Post install checks failed. Check
error.log\n";
        exit;
    }

    // show login dialog
    $this->_loginWindow =& new OWTLogin();
    $this->_loginWindow->show();

    /* Run the main loop. */
    $this->setState(
OWT_CONTROLLER_STATUS_LOGIN);
    Gtk::main();
}

function loginComplete($firstTime) {
    // notification when the login window
    // is closed
}

...
}

```

End listing 3

Powering up your GTK combo menus

For those of us coming from a mainly Web-programming environment (myself included), the behaviour of Gtk drop down lists, as implemented by GtkCombo, is not exactly what we would expect. For a start, you can choose from several different interactive behaviours using GtkList::set_selection_mode(), several of which are not possible in a Web browser. Furthermore, it turns out that a GtkCombo is actually made up of two pieces, a GtkEntry for the text entry/display field, and a GtkList for the drop-down list part.

The result of all this, is that there's a little bit of work to tame this combo box and make it easy to use. We use the class *SingleSelectWidget* for this, which provides a convenience wrapper around GtkCombo. The main useful methods provided by this class are:

- *function addOption(\$value, \$label, \$depth=0)* – use this to add an item to the menu
- *function getSelection()* – get the currently selected values

A similar convenience class *MultipleSelectWidget* is provided as a wrapper for a multiple selection list.

The off-line data store

One of the key approaches underpinning the design of this project was to replicate in the off-line client's SQLite database a subset of the tables from the on-line CMS system's PostgreSQL database. Our theory was that in doing this, we would be able to re-use unchanged data access code from our on-line CMS system.

In order to achieve this, we needed a solid replication mechanism. However, we had some pretty specific and unusual requirements, as follows:

- replicate from a PostgreSQL master to a SQLite slave
- support initial schema population in the SQLite client
- work over XML-RPC
- tight integration

There were a couple of blind alleys which I investigated before realising that I needed to draw on some existing and proven code. Thankfully, someone had walked this path before me. One of the

reasons I love working with open source is that I can take an almost-right solution and adapt it to my specific needs. This simply wouldn't be possible with a proprietary database system.

Introducing RServ

The almost-right solution which seemed most obvious to use was RServ. This is a standard contrib module in all PostgreSQL distributions, written back in 2000 by Vadim Mikheev and Thomas Lockhart. RServ is a master to multiple slave replication system written mainly in Perl, with a few key stored procedures (triggers) written in C.

It works by adding these triggers to tables which are to be replicated. Whenever a row is inserted, updated, or deleted in a synchronised table, the trigger causes a log entry to be added to a special logging table (`_rserv_log_`). The log table records the table id and primary key of the affected row.

When a synchronisation is to be performed, the `Replicate` script is run. This looks at the internal tables to determine what range of data needs to be synchronised for the given slave. It then generates a dump file from the data in the `_rserv_log_` table.

Customisation of RServ

As mentioned above, some customisation was needed in order to make use of RServ. In considering how to go about the customisation, the following factors weighed in:

- I've never been a great Perl hacker
- the customisations I planned required quite significant changes to RServ
- I wanted tight integration with the system

The result was that I decided to port the Perl part of RServ to PHP. The new PHP classes were as follows:

- `eZSyncProvider`: runs on the central server. Responsible for responding to a sync request by generating a gzip-compressed "sync dump" file.
- `eZSyncConsumer`: runs on the client. Responsible for reading in a sync dump file and using it to populate the local database.
- `eZSyncManager`: runs on the central server. Responsible for registration of the correct database tables for replication at installation time. Also handles registration and expiration of clients.

The hardest part of the port was the code for generation/parsing of the "wire format" data dump. These had to be 100% reliable even in boundary cases such as new lines appearing within text strings.

I was wary of trying to replicate RServ's custom log file format. Even though the format was basically tab separated values, there were subtleties about proper escaping of strings which looked like perfect ground for a few nasty bugs to hide away in. Instead, I used PHP's built in `serialize()` and `unserialize()` functions to dump data records. The resulting wire format is shown in listing 4.

The format contains commands `SYNCID`, `SCHEMA`, `UPDATE`, `DELETE` and `OK`, followed by any corresponding data. This is the same as the original Rserv format, except for the addition of the `SCHEMA` command. Notice that in the `UPDATE` data block, the first line is a number which is the length of the serialized string, followed by the string itself. This is done to handle the case when a string in a serialised structure might contain new lines.

Listing 4: Example wire format dump file

```
-- SYNCID 201
-- SCHEMA ezaddress_country
a:3:{s:14:"primaryKeyInfo";a:5:{s:
9:"tablename";s:17:"ezaddress_country";s:
18:"primarykeyattindex";s:1:"1";s:
14:"realprimarykey";s:2:"id";s:
17:"realprimarykeyoid";s:5:"18830";s:
14:"primarykeyname";s:2:"id";}s:6:"schema";a:8:
{i:1;a:2:{i:0;s:2:"id";i:1;s:4:"int4";}i:2;a:2:
{i:0;s:8:"parentid";i:1;s:4:"int4";}i:3;a:2:{i:
0;s:3:"iso";i:1;s:7:"varchar";}i:4;a:2:{i:0;s:
4:"name";i:1;s:7:"varchar";}i:5;a:2:{i:0;s:
6:"hasvat";i:1;s:4:"int4";}i:6;a:2:{i:0;s:
7:"removed";i:1;s:4:"int4";}i:7;a:2:{i:0;s:
```

```

9:"iscountry";i:1;s:4:"int4";}i:8;a:2:{i:0;s:
4:"iso3";i:1;s:7:"varchar";}s:7:"indices";a:0:
{}}
-- UPDATE ezaddress_country
115
a:8:{i:0;s:3:"830";i:1;s:3:"972";i:2;s:1:" ";i:
3;s:15:"Channel Islands";i:4;s:1:"0";i:5;s:
1:"0";i:6;s:1:"1";i:7;N;}
113
a:8:{i:0;s:1:"8";i:1;s:3:"974";i:2;s:2:"AL";i:
3;s:7:"Albania";i:4;s:1:"0";i:5;s:1:"0";i:6;s:
1:"1";i:7;s:3:"ALB";}
114
a:8:{i:0;s:2:"12";i:1;s:3:"957";i:2;s:2:"DZ";i:
3;s:7:"Algeria";i:4;s:1:"0";i:5;s:1:"0";i:6;s:
1:"1";i:7;s:3:"DZA";}
122
a:8:{i:0;s:2:"16";i:1;s:3:"966";i:2;s:2:"AS";i:
3;s:14:"American Samoa";i:4;s:1:"0";i:5;s:
1:"0";i:6;s:1:"1";i:7;s:3:"ASM";}
\.
-- UPDATE ezaddress_language
45
a:4:{i:0;s:1:"1";i:1;s:4:"Afar";i:2;N;i:3;N;}
50
a:4:{i:0;s:1:"2";i:1;s:9:"Abkhazian";i:2;N;i:
3;N;}
50
\.
-- OK
>>>end listing 4<<<

```

Client-server communication using XML-RPC

At this point we had a working replication mechanism which could fetch changed data from a PostgreSQL database and use it to populate a SQLite database. Now we needed to make this work robustly across an intermittent, low-bandwidth dial-up link.

We used the XML-RPC libraries provided with eZPublish 2 to do this. There is a tutorial on this at <http://zez.org/article/articleview/47/>, so I will not go into detail here. However, I will outline the key messages and responses used in the client-server communication:

Message	Parameters	Return
Login	Username, Password	Cookie ID/fail
Publish Article	Full Article Record	Remote Article ID on success, permission denied error on fail
Sync request	Client ID	URL and md5 checksum of dump package for subsequent download
Sync success	Client ID, Sync ID	Acknowledge

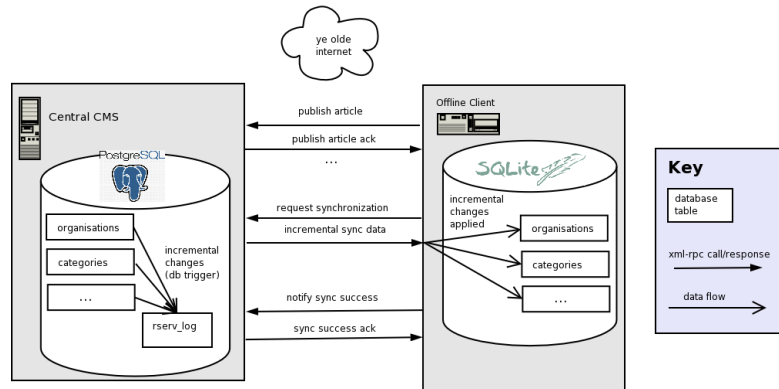
The diagram shown in Figure 2 gives an overview of the communication flow (time runs from top to bottom). Note: the login message is omitted from this diagram.

In order to check that our communication protocol is robust, we need to follow the flow of communication while assuming that the network connection will die at any given point in time. There are a few points where this could be particularly bad:

- After the “publish article” call but before the acknowledgement is received. There could be a danger that the client thinks the article did not publish successfully and tries again another time, resulting in the article being published twice. We work around this by giving the central system responsibility for mapping the (*ClientID*, *LocalArticleID*) pair onto the *RemoteArticleID*. Then if the client tries to re-publish the same article again, it will simply overwrite the already published article with the same data.
- After the “sync request” but before the response. In this case, we just waste CPU cycles on the central server generating a data dump which never gets used – unfortunate, but not a major problem.

- After the client has downloaded and applied the sync data, but before the “notify sync success” message can be sent. In this case, the client will get the same data set on a subsequent sync request. We circumvent this problem by using an insert-or-update approach for changed/inserted rows, and observe that if you try to delete the same row twice, you end up with the same end result. In other words, sync dumps can be applied repeatedly at the client with the same end result (this property is called *idempotence*).

Figure 2: System architecture and communication



Resumable downloads

Experiments showed that the initial data dump from the central system was approx 4MB of gzip-compressed data. Subsequent incremental downloads are much smaller, of the order of 40kbytes. Downloading 4MB of data over an unreliable dial-up line could pose a real challenge; it might even be impossible.

To get around this worry, we implemented a system of resumable downloads using *libcurl* and a handy feature of the HTTP 1.1 protocol called “ranged GETs.” This works by sending the *Content-Range* header along with your regular HTTP GET request, specifying the range of bytes that you need.

The resumable downloads are handled by a class called *eZOWtResumableDownload*. This is a “persistent” data object (i.e. it can be stored to the database), and a record is stored to the client’s SQLite database before any download attempt commences. This means that, even if the program crashes during download, the download can be resumed.

When a net cut-out does happen, the next user request to perform a sync operation will detect the incomplete download operation and attempt to resume it. The resumption algorithm looks at the length of the local, incomplete file and requests a resumption from that point. Listing 5 shows this. Once a download is completed, the local file is verified against the md5 checksum reported by the server, and finally the resumable download record is deleted from the DB.

Listing 5: Resumable download with cURL

```
class eZOWtResumableDownload {
    ...
}

/*!
Downloads data from $url to $localfile

This is implemented using the CURL libraries,
which are more robust than the built in fopen
wrappers. Also it will attempt to resume a
previous partial download using ranged GET
calls if needed.

\return
1 => on success
-1 => failed to open output file for writing
(permission denied?)
-2 => failed to open remote url for reading
```

```

-3 => failed during write to local file (disk
full?)
*/
function downloadWithCurl() {
    $localfile = $this->localFileName();
    $url = $this->getFullUrl($this->url);

    if (eZFile::file_exists($localfile)) {
        // resume
        $resumePoint = @eZFile::filesize(
            $localfile);
        $outputFileHandle = eZFile::fopen(
            $this->localFileName(), 'ab');
    }
    else {
        $resumePoint=0;
        $outputFileHandle = eZFile::fopen(
            $this->localFileName(), 'wb');
    }

    if (!$outputFileHandle) {
        return -1;
    }

    $this->_curlOutputFileHandle =
        $outputFileHandle;

    $ch = curl_init ($url);
    if (!$ch) {
        return -2;
    }

    if ($resumePoint>=$this->filesize) {
        $this->logMessage("Previous transfer is
complete. No more data to download. Continuing",
OWT_MESSAGE_INFO);
        return 1;
    }
    else if ($resumePoint) {
        $this->logMessage("Resuming previously
aborted transfer from byte $resumePoint",
OWT_MESSAGE_INFO);
        curl_setopt($ch, CURLOPT_RESUME_FROM,
$resumePoint);
    }

    curl_setopt($ch, CURLOPT_LOW_SPEED_LIMIT,
        20); // min bytes/sec
    curl_setopt($ch,CURLOPT_LOW_SPEED_TIME,
        500); // in secs
    curl_setopt($ch,CURLOPT_FAILONERROR,1);
    curl_setopt($ch,CURLOPT_WRITEFUNCTION,
        array(&$this, "curlWriteCallback"));

    $this->_downloadStartStamp=time();
    $this->_downloadedBytes=$resumePoint;

    $success = curl_exec($ch);

    if (!$success) {
        $this->logMessage('Transfer failed. Reason:
'.curl_error($ch), OWT_MESSAGE_ERROR);
        return -3;
    }

    if ($outputFileHandle) {
        unset($this->_curlOutputFileHandle);
        fclose( $outputFileHandle );
    }

    return 1;
}
...
}

```

>>end listing 5<<

Security issues

Security issues should be carefully considered. We store a local copy of the username + password hash in order to allow users to “log in” while off-line. If doing this, it's important to ensure passwords are strong enough against a brute force attack: a 5 character alphanumeric password is crackable on standard hardware in about 10 days. A 6 character password would take ~1.5 years.

A local login provides a moderate protection against a naïve attack from a local user who might gain physical access to an unattended computer with the aim of submitting unauthorised articles. Of course a more sophisticated attacker could always directly alter the SQLite database to queue unauthorised articles for publication (if they could get write access to the database file).

In its present implementation, the login message sends the username/password pair in plain text. There is room for improvement here – it would be good to change this to use SSL, which in theory should not be too difficult. On successful login, a random session cookie is generated, and this is used for subsequent authentication. The session is closed when the “sync success” message is sent in order to minimise the opportunity for session hijacking.

Reusing code from the Web CMS

One of the motivations in using PHP-GTK was that we might be able to re-use code from our Web-based CMS. There were a few candidates which we had in mind:

- data object classes
- XML-RPC library
- Logging
- SQL Interface

Things went mostly according to this plan. We organised the directory structure of the OWT as follows:

```
<OWTInstallDir>/
  lib/
  wwwlib/
```

- *wwwlib* contains relevant files taken verbatim from our CMS web app.
- *lib* contains OWT specific files or customised versions of files from the web app.

During initialisation, we set up PHP's *include_path* so that *lib* comes first, followed by *wwwlib*. This allows *lib* to override any files in *wwwlib*. We were able to retain most data object classes unmodified in *wwwlib*. However, some classes did need modification, mostly to strip out irrelevant functionality which introduced too many dependencies on other classes. Using the directory structure described, we could do this by simply adding a customised version of the class in a corresponding path under *lib*.

We also encountered one minor compatibility issue with SQLite. This is encountered when fetching query results as arrays, and is demonstrated in listing 6. This problem only appears to occur with specific versions of sqlite library: version 2.8.6 (which I have on my Linux machine) does not exhibit this behaviour, whereas 2.8.11 (on my Windows machine) does exhibit this behaviour. I believe that more recent versions of sqlite have added an option to toggle this behaviour (see <http://www.sqlite.org/cvstrac/chngview?cn=1250>). However, to ensure compatibility, we implemented a work around for this in our database layer which detects dot-qualified column names and converts them to the unqualified form.

One other setting affecting compatibility is *sqlite.assoc_case*. This can be set in *php.ini* or using *ini_set* at runtime, and offers three options for “case folding” of hash keys returned by *sqlite_fetch_array*: 0=>mixed case, 1=>upper case, 2=>lower case. To make sqlite look more like PostgreSQL, we set this to option 2 for lower case.

>>>Listing 6: incompatibility in some versions of SQLite<<<

```
<?php
```

```
$db = sqlite_open('mydb.sqlite', 0666,
  $sqliteerror);
```

```
sqlite_query($db, "create table test (a int)");
sqlite_query($db, "insert into test values
(1)");
$q=sqlite_query($db, "SELECT t.a from test t");
$tables = sqlite_fetch_array($q);

print_r($tables);

/*
Gives this unexpected output:
Array
(
    [0] => 1
    [t.a] => 1
)
*/
?>

>>>End listing 6<<<
```

Packaging it all up

Of course, if you really want people to use your application, it needs to be easy to install. Certainly you won't find users fiddling around trying to configure PHP-GTK! We tackled this by building an installer package using Nullsoft's Scriptable Installer System. This is an open source installer system. The installer package includes the full PHP runtime environment and all libraries needed by the system, meaning that the end user needs just a simple double-click to install the fully working system. We also used Apache Ant to automate the build process.

Evaluation and Conclusions

The off-line editor tool has only been in use for about 1 month, but we have already had some good feedback. My colleague Bornwell Mwewa, based at OneWorld Africa in Zambia, told me:

"It will reduce our idle time at the office when there is no connectivity. Usually we save and edit our stories from the Internet in Word and later publish them, but sometimes after you have done all this work you discover there is no connection. This will not hinder us now."

So far we have had no reports of trouble installing the software or of any stability issues or crashes. The latter in particular was something we have been looking out for, although it is still early and I would hesitate to give a guarantee that there will never be any stability issues – the jury is still out.

Another key aspect is usability and general user-friendliness of the tool. I'm quite pleased in this respect – PHP-GTK allows a much richer, more interactive user interface than a Web-based CMS. Things like trees, tabbed dialogs, and so on are a breeze to create, and the general user interaction can be much snappier when you don't need a server round trip on every click. One area for improvement would be the addition of a WYSIWYG editor for formatting of article text. This could be done either using the GtkHTML component, or possibly using GtkScintilla.

On the less positive side, one area which I haven't yet figured out is the extent to which PHP-GTK does or does not support Unicode.

The other major negative is the extent to which the current software is tied to OneWorld's CMS system. It would be nice if others were able to reuse the tool with their own CMS system, but at present there are several quite strong ties to OneWorld's customised version of eZPublish 2. In particular, I don't know how you could implement a similar custom replication mechanism for a CMS using MySQL (perhaps it would be possible to process MySQL's binary format replication logs).

In closing, I can heartily recommend PHP-GTK as an option for rapid development of desktop client applications. The benefits are particularly strong if you have an existing PHP Web application for which you need a desktop client tool, or if your team has a particular specialisation in PHP. Renewed interest in PHP-GTK since the release of PHP5 has re-invigorated community activities. This includes work on updated documentation, which should remove one of the biggest obstacles for newcomers. Come and join the fun!

Links & Literature

- Download the OWT source code from <http://davidheath.org/>
- OneWorld.net: <http://www.oneworld.net/>
- PHP-GTK website: <http://gtk.php.net/>
- PostgreSQL: <http://www.postgresql.org/>
- SQLite: <http://www.sqlite.org/>
- Glade/GTK for windows: <http://wingtk.sourceforge.net/>
- Nullsoft scriptable installer system: <http://nsis.sourceforge.net/>
- SQLite integration with Mozilla http://bugzilla.mozilla.org/show_bug.cgi?id=254886 and <http://www.mozilla.org/projects/sql>