The phpWatch 2.x.x Developer API

Aaron M. Rosenfeld

April 15, 2010

1 Introduction

This document provides developers with the information necessary to develop full featured, well integrated extensions for phpWatch 2. Nearly all of the externally available functionality is addressed in an attempt to educate developers about the full potential of their extensions.

2 Terminology

It is currently possible to create two types of extensions for phpWatch: monitor extensions and channel extensions. A *monitor extension* provides a new way of querying a service. For example, one could write a monitor that interacts with a service in a way unique to the specific service. This can be extremely powerful for services that may require more elaborate techniques to establish a connection or validate output.

Channel extensions provide a new way of notifying contacts when a service is found to be offline or non-functional. phpWatch comes with a number of basic channels, namely one that e-mails contacts and one that can text-message contacts. By developing a channel extension, it is possible to notify humans in other ways or to react to an outage automatically either with a PHP script alone or by invoking some outside process.

3 Getting Started

Developing extensions in phpWatch is designed to be simple but there is a slight learning curve. To get started, it is recommended that this document is read in its entirety. Then, use the `ConnectionMonitor` as an example monitor or `EmailChannel` as an example channel to learn from. Both are relatively simple.

4 Source Architecture

phpWatch is built using common object-oriented practices with source divided into two directories. The majority of the logic resides in the `src` directory, within which are two more directories: `monitors` and `channels`. All extensions go into one of these two sub-directories and extend either the `Monitor` or `Channel` class.

The second main source directory is `frontend`. This houses all files related to the display of the phpWatch frontend. Extension developers need only be concerned with the `frontend/forms` directory. Within this are two more sub-directories: `monitors` and `channels`. Much like the `src` directory, all extensions will include code in one of these directories.

It is *extremely* important to understand that some users run phpWatch without a frontend. That is, they manipulate the database through means other than the frontend. Therefore it is crucial for extension developers to understand which code goes within which directory. *All* code within the `src` directory must be entirely void of HTML. Nothing within this directory shall dictate the display capabilities of the system.

Conversely, as little logic as possible should be placed in `frontend` to allow other users to implement their own frontend or use scripts to manipulate information.

5 Developing an Extension

Extensions are comprised of *logical* code and *display* code. Logical code handles the manipulation of data, provides handles to the display code, and performs other actions specific to the extension. For monitor extensions, this code is placed in `src/monitors` and for channel extensions in `src/channels`. Display code goes in either `frontend/forms/monitors` or `frontend/forms/channels`. Such code is comprised mainly of HTML and function calls to some helper classes phpWatch provides.

5.1 Monitors

Monitor extensions provide a method of determining the status of services. Specifically, it handles the querying of a given service and discerning if the response indicates that the service is functioning properly (online) or improperly (offline).

5.1.1 Logic

All monitors extend the `Monitor` class within `src`. The new class *must* be placed in `src/monitors` in its own file named after the class. For example, to create a monitor named "ExampleMonitor", one would create the file `src/monitors/ExampleMonitor.php` containing a class `ExampleMonitor` which extends `Monitor`.

By design, monitors rarely interact with the database. This was done to reduce complexity and chances of corruption. To maintain information about a monitor, the class variable `config` is used. `$this->config` is accessible to monitor subclasses and is automatically saved to the database when the monitor is added or edited. Thus, if a monitor requires users to set a certain parameter, say a timeout period, the value of `$this->config['timeout']` may be used.

Extending `Monitor` also requires that the developer implement at least the following methods:

`public function queryMonitor()`	
Handles the actual querying of the monitor. This may entail opening a socket, sending a ping, etc. The function must return `true` or `false` if the monitor is considered online or offline respectively. This function is only called during cronjobs or manual re-querying.	
Parameters	None
Returns	`true` if monitor is online `false` if monitor is offline

`public function customProcessAddEdit($data, $errors)`	
This method is called when the form to add or edit a monitor is submitted. The `$data` array is simply the `$_POST` or `$_GET` array from the submission. All fields added by this monitor shall be validated in this method. If errors are encountered, append them to the `$errors` array, keyed by form-field name with the value of an error message string.	
Parameters	`$data`: Key-value array of form data. `$errors`: Array of errors. Append to this array if necessary.
Returns	Possibly modified `$errors` array.

`public function customProcessDelete()`	
This method is called when the form to delete a monitor is submitted. It is the invoked before built-in phpWatch functions that delete any information from the database and can therefore function with full querying capabilities.	
Parameters	None
Returns	None

`public function getName()`	
Gets the name of the monitor. This should return a short string such as "Connection Monitor" as it's used to populate form field such as drop down menus.	
Parameters	None
Returns	Short name of monitor as a string.

`public function getDescription()`	
Gets a description of the monitor. This should return a string that describes how the monitor queries services.	
Parameters	None
Returns	Description of monitor type.

5.2 Channels

Channel extensions provide a method of method of notification when a monitor is found to be offline. More technically, it dictates the set of actions to take when a monitor does not respond or malfunctions.

All channels extend the `Channel` class within `src`. The new class *must* be placed in `src/channel` in its own file named after the class. For example, to create a channel named "ExampleChannel", one would create the file `src/channels/ExampleChannel.php` containing a class `ExampleChannel` which extends `Channel`.

Like monitors, channels rarely interact with the database. To maintain information about a channel, the class variable `config` is used. `$this->config` is accessible to channel subclasses and is automatically saved to the database when the channel is added or edited. Thus, if a channel requires users to set a certain parameter, say an e-mail address, the value of `$this->config['email_address']` may be used.

Additionally, extending `Monitor` requires that the developer implement at least the following methods:

`public function doNotify()`	
Called when a notification should be sent. This method shall invoke custom code which sends a notification specific to the channel.	
Parameters	`$monitor`: A reference to the monitor object that was found to be offline.
Returns	None

`public function customProcessAddEdit($data, $errors)`	
This method is called when the form to add or edit a channel is submitted. The `$data` array is simply the `$_POST` or `$_GET` array from the submission. All fields added by this channel shall be validated in this method. If errors are encountered, append them to the `$errors` array, keyed by form-field name with the value of an error message string.	
Parameters	`$data`: Key-value array of form data. `$errors`: Array of errors. Append to this array if necessary.
Returns	Possibly modified `$errors` array.

`public function customProcessDelete()`	
This method is called when the form to delete a channel is submitted. It is the invoked before built-in phpWatch functions that delete any information from the database and can therefore function with full querying capabilities.	
Parameters	None
Returns	None

`public function getName()`	
Gets the name of the channel. This should return a short string such as "E-mail Channel" as it's used to populate form field such as drop down menus.	
Parameters	None
Returns	Short name of channel as a string.

`public function getDescription()`	
Gets a description of the channel. This should return a string that describes how the channel notifies contacts.	
Parameters	None
Returns	Description of channel type.

5.3 Display

Display code for monitors and channels both consists of a form that allows for modification of `$config` values. As in the example above, if there is an entry in `$this->config['email_address']`, there will be a form field for "E-mail Address". It's important to note that fields common to all monitors (hostname, port, etc.) are generated automatically along with the form start/stop HTML. Channels have no user-editable fields and therefore only the form start/stop HTML is automatically generated.

To create a form, a file named exactly the same as the monitor or channel must be placed in `frontend/forms/monitor` or `frontend/forms/channel` respectively. For example, to create a form for a monitor named `ExampleMonitor` create the file `frontend/forms/monitor/ExampleMonitor.php`. Field generation is made simple with phpWatch. The `FormHelpers` class provides methods to create most HTML field types along with facilities for error handling. A detailed summary of each method isn't shown but all fields except select boxes can be created with

```
FormHelpers::createTYPE($name, $value, $attribs = null)
```

where `TYPE` should be replaced with a field type such as `Text`, `Hidden`, `Checkbox`, etc.

Select boxes can be created in a similar way with the only difference being the second parameter is now an array of options, each generated with `FormHelpers::getOption($display, $value, $attribs = null)`.

See `frontent/FormHelpers.php` for more information on form creation and `frontend/forms/monitors/ConnectionMonitor.php` for an example.

6 Considerations

Developing an extension for phpWatch was designed to be as simple as possible while still placing a great deal of control in the hands of developers. However, with this control comes the ability to cause error. In particular, monitor query code and channel notification code that fails either due to syntax error or exceptions will likely cause a failure of cronjobs.

7 When Not to Develop an Extension

The developers of phpWatch want others to develop extensions. In fact, the *majority* of development time was spent assuring others could quickly learn and integrate into phpWatch. However, it is extremely important to realize what should and what shouldn't be its own extension.

To make this decision, one must ask what is unique about the desired monitor or channel. If you find yourself making extensions that vary only in terms of parameters, that is a poor extension. For example, making a monitor that waits 4 seconds for a server response and then making one that waits for a 10 second response is unnecessary. Make a monitor that waits n seconds instead.

Although a trivial example, it is important to understand. A good indicator of a poor monitor or channel is one with nothing being set in the `$this->configs` array. This indicates that the monitor serves exactly one purpose with no flexibility. Although there are times this is acceptable, many times it is not.