By Vikram Vaswani

Table of Contents

Getting Real

The nice thing about the open−source community is that no matter how arcane your requirement, there's usually someone out there who's got a tool to help you meet it. It might not be the best tool in the world, it might be a little rough around the edges, it might not even look very pretty, but it'll usually get the job done. And you won't pay a cent for it.

Take, for example, the task of manipulating application configuration data. You know what I mean − asking the user for configuration values at install time, saving this data to a file, and using it when required at run time. It's a pretty simple exercise, and one that's fairly standard across every application − and, like me, you've probably done it a few hundred times over the last couple years without even thinking about it.

Unlike me, though, you probably had the good sense to create a library of reusable functions to help you accomplish this task quickly, and with minimal effort. If you did, you probably don't need to read any further; you can log off and catch some zzzzs instead, since you're obviously a Real Programmer, and everyone knows that Real Programmers need their beauty sleep...

If, on the other hand, you still handcraft the code to manage your configuration data every time you build an application, you're definitely going to want to read this article − it's your first step on the road to Real Programmer−hood. Flip the page, and let me tell you all about patConfiguration.

Plug And Play

patConfiguration is a PHP−based tool designed, in the author's words, to "access XML−based configuration files via PHP". Developed by Stephan Schmidt, it is freely available for download and is packaged as a single PHP class which can be easily included in your application.

Very simply, patConfiguration provides application developers with a set of APIs that ease the task of reading, writing and maintaining application configuration files. As a tool designed to assist in the manipulation of data, it fully supports the XML data markup toolkit, and is capable of producing configuration files in both XML and PHP format. It supports a variety of different data types for configuration values, comes with the ability to link configuration variables together, and supports caching of configuration data for better performance.

If you're in the business of building Web applications, and if those applications require some amount of configuration to get up and running, you're going to find patConfiguration invaluable to your development cycle. Written as a PHP class, patConfiguration can easily be integrated into any PHP−based Web application, and can substantially reduce the amount of time you spend manipulating application configuration files and data. You'll find it functional, powerful and (if you're the kind who likes fiddling) easily extensible, and it'll soon be a standard component of every application you write.

Before proceeding further, you should visit the patConfiguration home page athttp://www.php−tools.de/and download a copy of the latest version (1.3 at the time of writing). The package contains the main class file, documentation outlining the exposed methods and variables, and some example scripts.

Your Friendly Neighbourhood Spiderman

Now that the hard sell is over and you're (hopefully) all set up with patConfiguration, let's take a simple example to see how it works.

Consider the following simple configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<path name="mail">
<configValue name="from-name" type="string">Peter
Parker</configValue>
<configValue name="from-address"
type="string">spider.man@roof.top.com</configValue>
</path>
</configuration>
```

The XML file above contains configuration data in a format that is understood by patConfiguration. As you can see, a patConfiguration-compliant configuration file must conform to the standard rules of XML markup, and must contain a <configuration> root element. Configuration values can be grouped together under this root element using <path> elements, with every variable-value pair represented by a <configValue> element.

Variable-value pairs are accessed by drilling down the tree of <path> elements until the desired node is reached. For example, to access the value of the email address

```
spider.man@roof.top.com
```

in the configuration file above, I would use the path

```
mail.from-address.
```

patConfiguration allows you to nest <path> elements to any depth - the following is a perfectly valid configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<path name="application">
<configValue name="name"
type="string">SomeApp</configValue>
<configValue name="version"
type="string">2.3</configValue>
```

```
<path name="window">
<configValue name="height"
type="int">600</configValue>
<configValue name="width"
type="int">500</configValue>
<path name="list">
<configValue name="maxItems"
type="int">5</configValue>
</path>
</path>
</path>
</configuration>
```

You can even link the values in a configuration file with each other via the <getConfigValue> element − consider the following example, which uses the application name and version number to dynamically create a variable containing the window title:

```
<?xml version="1.0" encoding="UTF−8"?>
<configuration>
<path name="application">
<configValue name="name"
type="string">SomeApp</configValue>
<configValue name="version"
type="string">2.3</configValue>
<path name="window">
<configValue name="height"
type="int">600</configValue>
<configValue name="width"
type="int">500</configValue>
<configValue name="title" type="string">
<getConfigValue path="application.name"
/> <getConfigValue path="application.version" />
</configValue>
</path>
</path>
</configuration>
```

The variable−value pairs in this configuration file can be read and manipulated by patConfiguration in the context of a PHP application. Let's look at that next.

Anatomy Class

Next, it's time to use the patConfiguration engine to read and use the values in the configuration file. Here's how:

```
<?

// include class
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf->setConfigDir("config");

// read config file
$conf->parseConfigFile("config.xml");

// print configuration
print_r($conf->getConfigValue());

?>
```

Here's the output:

```
Array
(
[application.name] => SomeApp
[application.version] => 2.3
[application.window.height] => 600
[application.window.width] => 500
[application.window.list.maxItems] => 5
)
```

Let's dissect this a little to see how it works.

1. The first step is, obviously, to include all the relevant files for the class to work.

```
// include class
require("patConfiguration.php");
```

Once that's done, I can safely create an object of the patConfiguration class.

```
// create patConfiguration object
$conf = new patConfiguration;
```

This object instance will serve as the primary access point to the data in the application configuration file(s), allowing me to do all kinds of nifty things with it.

2. Next, the object's setConfigDir() method is used to set the default location of the configuration files,

```
// set config file locations
$conf->setConfigDir("config");
```

and the parseConfigFile() method is used to actually read each file into the object's internal stack.

```
// read config file
$conf->parseConfigFile("config.xml");
```

You can parse multiple configuration files by calling parseConfigFile() for each file, and telling patConfiguration to append (instead of overwriting) each set of configuration variables to the existing stack via the additional "a" option − as in the following code snippet:

```
// read config files
$conf->parseConfigFile("config.main.xml");
$conf->parseConfigFile("config.local.xml", "a");
$conf->parseConfigFile("config.users.xml", "a");
```

4. Finally, all that's left is to actually use the configuration data − in this case, print it all to the standard output device.

```
// print configuration
print_r($conf->getConfigValue());
```

The getConfigValue() method gets the value of a specified configuration variable from the configuration file(s). If no variable name is specified, the entire set of values is returned...as in the example above.

Version Control

Let's now consider a variant of the example on the previous page, this one printing the value of a single configuration variable:

```
<?

// include class
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf->setConfigDir("config");

// read config file
$conf->parseConfigFile("config.xml");

// get configuration values
print_r($conf->getConfigValue("application.version"));

?>
```

In this case, since the getConfigValue() method receives the path and name of the configuration value to be retrieved, only that value is retrieved and printed.

Here's a more realistic example, this one using an XML configuration file containing MySQL database access parameters

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<path name="mysql">
<configValue name="host"
type="string">localhost</configValue>
<configValue name="user" type="string">joe</configValue>
<configValue name="pass"
type="string">secret</configValue>
<configValue name="db" type="string">db456</configValue>
</path>
</configuration>
```

to open up and use a database connection.

```
<?

// include class
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf->setConfigDir("config");

// read config file
$conf->parseConfigFile("config.xml");

// get and use configuration values
$connection = mysql_connect(
$conf->getConfigValue("mysql.host"),
$conf->getConfigValue("mysql.user"),
$conf->getConfigValue("mysql.pass")
) or die ("Unable to connect!");

mysql_select_db(
$conf->getConfigValue("mysql.db")
) or die ("Unable to select database!");

$query = "SELECT * FROM users";

$result = mysql_query($query) or die ("Error in query: $query.
" .
mysql_error());

mysql_close($connection);

?>
```

The Write Stuff

So that takes care of reading configuration files − now how about writing them?

In the patConfiguration universe, creating a configuration file is a two−step process: first add one or more entries (variable−value pairs) to the configuration stack, and then write the stack to a file. patConfiguration comes with a setConfigValue() method that takes care of the first step, and a writeConfigFile() method that takes care of the second one. Consider the following example, which demonstrates:

```
<?

// include class
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf->setConfigDir("config");

// set config values
$conf->setConfigValue("screen.width", 500);
$conf->setConfigValue("screen.height", 650);

// write file
$conf->writeConfigFile("config.xml", "xml");

?>
```

The setConfigValue() method accepts two primary parameters − variable name and corresponding value − and a third optional parameter, which is the datatype of the variable being set (if this third value is omitted, patConfiguration will automatically determine the variable type). Valid type values include "string", "integer", "boolean", "float" and "array" − I'll be discussing these again a little further down.

Once the variable−value pairs have been created, the writeConfigFile() method is called to actually write the configuration data to a file. In addition to the file name, patConfiguration also allows you to specify the file format − "xml" or "php" − together with a couple of miscellaneous options that adjust the behaviour of the writeConfigFile() method.

Here's the output of the example above:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<configuration> <path name="screen"><configValue name="height"
type="int">650</configValue><configValue name="width"
type="int">500</configValue> </path></configuration>
```

Having trouble reading it? Have patConfiguration indent it a little more neatly with the additional "mode" parameter:

```
<?

// include class
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf->setConfigDir("config");

// set config values
$conf->setConfigValue("screen.width", 500);
$conf->setConfigValue("screen.height", 650);

// write file
$conf->writeConfigFile("config.xml", "xml", array("mode" =>
"pretty"));

?>
```

Here's the revised output:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<configuration>
<path name="screen">
<configValue name="height" type="int">650</configValue>
```

```
<configValue name="width" type="int">500</configValue>
</path>
</configuration>
```

You can set multiple configuration values at a time via the setConfigValues() method, which accepts an array of variable−value pairs

```
<?

// include class
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf−>setConfigDir("config");

// create array of configuration values
$config = array(
"screen.height" => 650,
"screen.width" => 500
);

// add to configuration table
$conf−>setConfigValues($config);

// write file
$conf−>writeConfigFile("a.xml", "xml");

?>
```

and clear a value using the clearConfigValue() method.

```
<?

// include class
```

```
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set and display value
$conf->setConfigValue("name", "Superman");
echo $conf->getConfigValue("name");

// clear and check to see if value exists
$conf->clearConfigValue("name"); echo
$conf->getConfigValue("name");

?>
```

Note that if no path is provided to clearConfigValue(), the entire configuration table is cleared.

Speaking Native

If XML isn't really your cup of tea, patConfiguration can also create configuration files using traditional PHP as well − this can work better in some cases, since the configuration file only needs to be include()d in your application, not parsed by an XML parser as well. Consider the following example and its output, which demonstrate how this works:

```
<?

// include class
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf−>setConfigDir("config");

// set config values
$conf−>setConfigValue("screen.width", 500);
$conf−>setConfigValue("screen.height", 650);

// write file
$conf−>writeConfigFile("config.php", "php");

?>
```

Here's the output:

```
<?PHP
// Configuration generated by patConfiguration

$config = array();
$config["screen.height"] = 650;
$config["screen.width"] = 500;
?>
```

As you can see, the variable value pairs defined in the patConfiguration table get converted into a PHP associative array and written to a file. By default, this associative array is named $config − you can change it by providing an alternative name via the "varname" argument to writeConfigFile().

```
<?

// include class
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf−>setConfigDir("config");

// set config values
$conf−>setConfigValue("screen.width", 500);
$conf−>setConfigValue("screen.height", 650);

// write file
$conf−>writeConfigFile("config.php", "php", array("varname" =>
"appConfig"));

?>
```

Here's the revised output:

```
<?PHP
// Configuration generated by patConfiguration

$appConfig = array();
$appConfig["screen.height"] = 650;
$appConfig["screen.width"] = 500;
?>
```

Not Your Type

You'll remember, from a couple pages back, that patConfiguration supports a number of different data types for its configuration variables. This variable type can be set via the setConfigValue() method, and currently can be any one of "string", "integer", "boolean", "array" and "float".

Here's an example which demonstrates how strings, integers and Booleans work:

```
<?

// include class
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf->setConfigDir("config");

// set config values
$conf->setConfigValue("screen.width", 500, "integer");
$conf->setConfigValue("font.face", "Verdana", "string");
$conf->setConfigValue("window.toolbar.visibility", true,
"boolean");

// write file
$conf->writeConfigFile("config.xml", "xml", array("mode" =>
"pretty"));

?>
```

Here's the XML output:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<configuration>
<path name="font">
<configValue name="face"
type="string">Verdana</configValue>
</path>
```

```
<path name="screen">
<configValue name="width" type="int">500</configValue>
</path>
<path name="window">
<path name="toolbar">
<configValue name="visibility"
type="bool">true</configValue>
</path>
</path>
</configuration>
```

Wanna use arrays in your configuration? patConfiguration supports those as well:

```
<?

// include class
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf->setConfigDir("config");

// set config values
$conf->setConfigValue("friends", array("Rachel", "Ross",
"Monica",
"Joey", "Chandler", "Phoebe"), "array");

// write file
$conf->writeConfigFile("config.xml", "xml", array("mode" =>
"pretty"));

?>
```

Here's the XML output,

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<configuration>
<configValue name="friends" type="array">
<configValue type="string">Rachel</configValue>
<configValue type="string">Ross</configValue>
<configValue type="string">Monica</configValue>
<configValue type="string">Joey</configValue>
<configValue type="string">Chandler</configValue>
<configValue type="string">Phoebe</configValue>
</configValue>
</configuration>
```

and here's the PHP output:

```
<?PHP
// Configuration generated by patConfiguration

$config = array();
$config["friends"] = array();
$config["friends"][0] = "Rachel";
$config["friends"][1] = "Ross";
$config["friends"][2] = "Monica";
$config["friends"][3] = "Joey";
$config["friends"][4] = "Chandler";
$config["friends"][5] = "Phoebe";
?>
```

When Time Is Money, Recycle!

Here's another example, this one demonstrating how patConfiguration can be used in the context of a script accepting user input for application configuration. This script is divided into two parts: a form which displays the current configuration (if available) and allows the user to edit it, and a form processor, which accepts the new configuration and saves it to a file.

In addition to patConfiguration, this script also uses the patTemplate engine for the actual interface generation − you can read more about patTemplate at http://www.devshed.com/Server_Side/PHP/patTemplate/

Here's the template,

```
<patTemplate:tmpl name="form">
<html>
<head><basefont face="Arial"></head>
<body>
<h2>Configuration</h2>
<table border="0" cellspacing="5" cellpadding="5">
<form action="configure.php" method="post">

<tr>
<td>MySQL host name</td>
<td><input type="text" name="db_host" value="{DB_HOST}"></td>
</tr>

<tr>
<td>MySQL user name</td>
<td><input type="text" name="db_user" value="{DB_USER}"></td>
</tr>

<tr>
<td>MySQL user password</td>
<td><input type="text" name="db_pass" value="{DB_PASS}"></td>
</tr>

<tr>
<td colspan="2" align="center"><input type="submit"
name="submit"
value="Save Configuration"></td> </tr>

</form>
</table>

</body>
</html>
</patTemplate:tmpl>
```

and here's the script that does all the work:

```
<?

// include classes
require("patConfiguration.php");
require("patTemplate.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf->setConfigDir("config");

// create patTemplate object
$template = new patTemplate;

// set template location
$template->setBasedir("templates");

// add templates to the template engine
$template->readTemplatesFromFile("configuration.tmpl");

// form not yet submitted
if (!$_POST["submit"])
{
// check to see if config file exists
if (file_exists("config/config.xml"))
{
// parse it and display configuration values
$conf->parseConfigFile("config.xml");
$template->AddVar("form", "DB_HOST",
$conf->getConfigValue("db.host"));
$template->AddVar("form", "DB_USER",
$conf->getConfigValue("db.user"));
$template->AddVar("form", "DB_PASS",
$conf->getConfigValue("db.pass"));
}

// parse and display the template
$template->displayParsedTemplate("form");

}
else
{
// accept the submitted values
// and write them to a configuration file
$conf->setConfigValue("db.host", $_POST['db_host']);
$conf->setConfigValue("db.user", $_POST['db_user']);
$conf->setConfigValue("db.pass", $_POST['db_pass']);
$conf->writeConfigFile("config.xml", "xml", array("mode" =>
"pretty"));

}
?>
```

In this case, patConfiguration is used to read the application's database configuration from a file via the parseConfigFile() method and display the variable−value pairs contained within that file in an editable HTML form. The user may then modify these values and submit the form; patConfiguration will accept the new values and write them back to the configuration file via writeConfigFile().

This kind of application configuration is pretty common to most Web−based tools − and patConfiguration lets you build an interface around it quickly and efficiently, with maximum code reuse and minimal time wastage.

Cache Cow

Normally, patConfiguration parses your XML configuration file every time you need to load configuration data; this is expensive, in terms of both time and processor cycles. In these situations, you can obtain a performance improvement by using patConfiguration's built−in cache, which uses a serialized representation of the XML data to speed up load time.

Using a cache is pretty easy − consider the following example, which demonstrates how:

```
<?

// include class
require("patConfiguration.php");

// create patConfiguration object
$conf = new patConfiguration;

// set config file locations
$conf−>setConfigDir("config");

// set cache locations
$conf−>setCacheDir("cache");

// read config file
$conf−>loadCachedConfig("config.xml");

// get configuration values
print_r($conf−>getConfigValue("application.version"));

?>
```

Using the cache is thus simply a matter of setting a cache directory and using the loadCachedConfig() method instead of the parseConfigFile() method. If the file does not already exist in the cache, patConfiguration will locate it, read and serialize it, and use the serialized version in all subsequent calls. When the data in the original file changes, this change is detected by patConfiguration and the cache is updated with the new data.

Finally, patConfiguration also comes with a number of extensions to simplify integration with, and configuration of, other pat classes − extensions are currently available for the patTemplate, patUser and patDbc classes, and the source distribution comes with numerous examples of how they may be used. I'm not going to get into the details here, but you should certainly take a look at this once you're comfortable with the basics of using patConfiguration.

Link Zone

That's about it for the moment. In this article, I introduced you to the patConfiguration class, which is designed primarily to assist you in the reading, writing and manipulation of configuration data. I showed you how to read and write configuration files in both XML and PHP format, and how to use patConfiguration's built−in methods simplify and speed up configuration tool development, and improve performance by caching frequently−accessed configuration data.

If you'd like to learn more about patConfiguration and its related tools, you should take a look at the following links:

The official patConfiguration Web site, at http://www.php−tools.de/

Web−based application development with patTemplate, at http://www.devshed.com/Server_Side/PHP/patTemplate/

patConfiguration API documentation, at http://www.php−tools.de/site.php?&file=patConfigurationDocApi.xml

Extending patConfiguration, at http://www.php−tools.de/site.php?&file=patConfigurationDocExtensions.xml

Till next time...ciao!

Note: All examples in this article have been tested on Linux/i586 with PHP 4.2.3, Apache 1.3.23 and patConfig 1.3. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!