Savant 2 API Reference

Taken from **www.phpsavant.com**

pmjones [AT] ciaweb [DOT] net

addPath

```
void addPath ( string type, string directory )
```

Security Note: As with all PHP functions that access the file system, you should be careful to check and/or sanitize any variables to you pass to this method, especially if they are accepted from an untrusted source (such as $_GET, $_POST, and $_REQUEST variables).

Adds a directory to the current search path. The type parameter is always 'template' or 'resource', and the directory parameter is the directory to add to the search path for the specified type.

For example:

```php
<?php
$tpl =& new Savant2();

// add a template directory
$tpl->addPath('template', '/path/to/templates');

// add a resource directory
$tpl->addPath('resource', '/path/to/resources');
?>
```

The last directory added is the first directory searched. That is, each time you call addPath(), that directory is added to the beginning of the search path.

```php
<?php
$tpl =& new Savant2();

// add a series of template directories
$tpl->addPath('template', '/path/one');
$tpl->addPath('template', '/path/two');
$tpl->addPath('template', '/path/three');

/*
Now when you fetch() or display() a template, Savant will
search in this order:

    /path/three
    /path/two
    /path/one
*/
?>
```

assign

Use assign() to assign copies of variables to a Savant template. There are three forms of the assign() method: assign by name, by array, or by object properties.

> *Note: In general and by convention, you should not assign variable names that start with an underscore ('_var', '_etc'). This is because Savant2 uses underscore-prefixed properties as private properties; this aids in identifying what variables to extract at display() and fetch() time.*

assign by name

void|Savant _ Error assign (string variable name, mixed variable value)

In the first form, the assign() method takes value of the second parameter and creates a template variable with the name of the first parameter. For example, your logic script might do this...

```php
<?php
$tpl =& new Savant2();

// creates a template variable called $title
// with a value "My Page Title"
$tpl->assign('title', 'My Page Title');

// this does the same thing
$string = 'My Page Title';
$tpl->assign('title', $string);

// so does this
$var = 'title';
$value = 'My Page Title';
$tpl->assign($var, $value);
?>
```

...so your template script could do this:

```html
<html>
    <title><?php echo $this->title ?></title>
</html>
```

direct assignment

Because Savant2 uses an object-oriented approach, you can assign variables by name directly to the object without using the assign() method at all:

```php
<?php
$tpl =& new Savant2();
$tpl->title = 'My Page Title';
?>
```

assign from associative array

void|Savant _ Error assign (array variable names and values)

In the second form, the assign() method takes one parameter: an associative array where the key is the name of the template variable to be created, and the value is the value for that template variable.

```php
<?php
$tpl =& new Savant2();

$array = array(
    'title' => 'My Page Title',
    'var1' => 'some value',
    'var2' => 'some other value',
    'var3' => 'yet another value',
    'books' => array(
        array(
            'author' => 'Stephen King',
            'title' => 'The Stand'
        ),
        array(
            'author' => 'Neal Stephenson',
            'title' => 'Cryptonomicon'
        ),
        array(
            'author' => 'Milton Friedman',
            'title' => 'Free to Choose'
        )
    )
);
$tpl->assign($array);
?>
```

This form of assign() makes it very easy to use the results of a PEAR DB::getAll() call; for example, to output an entire table of results. The logic script...

```php
<?php
```

```php
$tpl =& new Savant2();

// assume $DB is the PEAR DB object, and that $savant is a
// Savant object.

$sql = 'SELECT author, title FROM books ORDER BY author, title';
$rows = $DB->getAll($sql);

if (! PEAR::isError($rows)) {
    $tpl->assign('books', $rows);
}

// display a template with the assigned variables.
$tpl->display('booklist.tpl.php');
?>
```

... and the associated template file:

```html
<html>
    <head>
        <title>Booklist</title>
    </head>
    <body>

        <?php if (isset($this->books) && is_array($this->books)): ?>
            <table>
                <tr>
                    <th>author</th>
                    <th>title</th>
                </tr>
                <?php foreach ($this->books as $key => $val): ?>
                    <tr>
                        <td><?php echo $val['author'] ?></td>
                        <td><?php echo $val['title'] ?></td>
                    </tr>
                <?php endforeach; ?>
            </table>

        <?php else: ?>

            No books in the list.

        <?php endif; ?>
    </body>
</html>
```

assign from object properties

void|Savant _ Error **assign** (object object instance)

In the third form, the assign() method takes one parameter: an object instance. All available object properties (object variables) will be assigned to the template.

> *Note: This does not assign the object itself, only the properties of the object; thus, the object methods will not be available within the template. Use assignRef() to assign the object itself by reference, or use the first form of assign() to assign a copy of the object, and then the object methods will be available within the template.*

```php
<?php
$object = new StdClass();

$object->title = 'My Page Title';
$object->var1 = 'some value',
$object->var2 = 'some other value',
$object->var3 = 'yet another value',

$object->books = array(
    array(
        'author' => 'Stephen King',
        'title' => 'The Stand'
    ),
    array(
        'author' => 'Neal Stephenson',
        'title' => 'Cryptonomicon'
    ),
    array(
        'author' => 'Milton Friedman',
        'title' => 'Free to Choose'
    )
);

$tpl->assign($object);
?>
```

assignRef

boolean|Savant_Error **assignRef** (string variable name, &mixed variable value)

This works just like the first form of assign() (i.e., assign by name) except it assigns a variable by reference, not by copy.

> *Note: In general and by convention, you should not assign variable names that start with an underscore ('_var', '_etc'). This is because Savant2 uses underscore-prefixed properties as private properties; this aids in identifying what variables to extract at display() and fetch() time.*

This is useful when you need to assign variables that will be manipulated by the template, and you want your business logic to have access to those changes automatically. That is, changes to the variable in the template will be reflected in your business logic (because the template is working with a reference to the original variable, not a copy of the variable).

```php
<?php
$tpl->assignRef('refvar', $referenced_variable);
?>
```

You can also assign references directly to the Savant object:

```php
<?php
$tpl->refvar =& $referenced_variable;
?>
```

clear

```
void clear ( [mixed vars default null] )
```

Clears the value of (i.e., unsets) an assigned variable. If the vars parameter is null, clears all assigned variables. If vars is a string, clears that one variable. If vars is an array, clears each variable named in the array values.

```php
<?php
$tpl =& new Savant2();

// clear all assigned variables
$tpl->clear();

// clear the $var assigned variable
$tpl->clear('var');

// clear these three assigned variables
$array = array('var1', 'var2', 'var3');
$tpl->clear($array);
?>
```

Constructor

```
object Savant2 ( [ array options default null ] )
```

This constructs a new Savant2 object.

```php
<?php
require_once 'Savant2.php';
$tpl =& new Savant2();
?>
```

You may pass an array of options as the only parameter.

You may use any or all of the following options:

Option Key	Description	Default
template_path	a path string (or array of directories) to search for templates	null
resource_path	a path string (or array of directories) to search for plugin, filter, and error classes	null
error	the string name of the Savant_Error class resource to use for errors	null
extract	boolean, whether or not to extract assigned variables at fetch() and display() time	false
template	string, the name of the default template to use for display() and fetch()	null

For example:

```php
<?php
require_once 'Savant2.php';

$opts = array(
    'template_path' => '/path/one:/path/two',
    'resource_path' => array('/path/three', '/path/four'),
    'error' => 'pear',
    'extract' => true,
    'template' => 'default.tpl.php'
);

$tpl =& new Savant2($opts);
?>
```

display

```
void|Savant _ Error display ( [ string source default null ] )
```

> *Security Note: As with all PHP functions that access the file system, you should be careful to check and/or sanitize any variables to you pass to this method, especially if they are accepted from an untrusted source (such as $_GET, $_POST, and $_REQUEST variables).*

Use the display() method to run a template source using the values you have assigned to a Savant object and then output the display to the browser.

The source parameter specifies which template source should be used; if null, display() will use the template source from setTemplate().

Either way, Savant will search for that source in the template search path. If Savant can not find the file, it will return a Savant_Error object. Savant will also (optionally) compile the source into a PHP script, if a compiler object has been set in Savant. Finally, all loaded filters will be applied to the template output.

```php
<?php
//  this displays a template from the template search path.
$tpl->display('template.tpl.php');

// this would display another template from a subdirectory
// in the template search path.
$tpl->display('subdir/anotheryawiki _ storelate.tpl.php');

// an alternative method: set the default template
// and then display the default.
$tpl->setTemplate('template.tpl.php');
$tpl->display();
?>
```

fetch

```
string|Savant _ Error fetch ( [ string source default null ] )
```

> *Security Note: As with all PHP functions that access the file system, you should be careful to check and/or sanitize any variables to you pass to this method, especially if they are accepted from an untrusted source (such as $_GET, $_POST, and $_REQUEST variables).*

Use the fetch() method to run a template source using the values you have assigned to a Savant object and then return the output to the calling script.

The script parameter specifies which template source should be used; if null, fetch() will use the template source from setTemplate().

Either way, Savant will search for that source in the template search path. If Savant can not find the file, it will return a Savant_Error object. Savant will also (optionally) compile the source into a PHP script, if a compiler object has been set in Savant. Finally, all loaded filters will be applied to the template output.

```php
<?php
//  this gets the template output from the template search path.
$output = $tpl->fetch('template.tpl.php');

// this would display another template from a subdirectory
// in the template search path.
$output = $tpl->fetch('subdir/anotheryawiki _ storelate.tpl.php');

// an alternative method: set the default template
// and then return the default script output.
$tpl->setTemplate('template.tpl.php');
$output = $tpl->fetch();
?>
```

getPath

```
array getPath ( [ string type default null ] )
```

Gets the a directories for a search path. The type parameter is always 'template' or 'resource'; if null, returns both paths as separate array elements.

getVars()

```
mixed getVars ( [ mixed vars ] default null )
```

Returns the names and values of assigned variables. If the vars parameter is null, returns an array of all assigned variables. If vars is a string, returns the value of that variable. If vars is an array, returns an array of all variable names and values named in that array.

```php
<?php
$tpl =& new Savant2();

// get all assigned variables
$vars = $tpl->getVars();

// get the $var assigned variable
$vars = $tpl->getVars('var');

// get these three assigned variables
$array = array('var1', 'var2', 'var3');
$vars = $tpl->getVars($array);
?>
```

Because Savant2 stores assigned variables as object properties, you can use get_object_vars to get variables as well.

```php
<?php
$tpl =& new Savant2();
$vars = get _ object _ vars($tpl);
?>
```

Note: This will return not only assigned variables but also the private properties of Savant.

isError

```
bool isError ( object var )
```

Tests if the var parameter is a Savant_Error object, or a subclass of a Savant_Error object.
Use this to test the return of methods such as assign(), display(), etc. to see if there were errors.
For example, to check if a template was successfully located and fetched:

```php
<?php
$tpl =& new Savant2();

$result = $tpl->fetch('template.tpl.php');

if ($tpl->isError($result)) {
    // error!
    echo "Problem displaying template.tpl.php";
    var_dump($result);
} else {
    // success!
    echo $result;
}
?>
```

loadFilter

```
void|Savant _ Error loadFilter ( string filterName [, array options default null [,
bool savantRef default false ] ] )
```

Loads a filter to be applied to all template output from display() and fetch().

If the filterName cannot be found in the resource search path, returns a Savant_Error.

The options array is used to pass options into the filter class.

If you set savantRef to true, a reference to the calling Savant object will be passed to the filter object.

Filters are applied in the order they are loaded.

```php
<?php
$tpl =& new Savant2();

$tpl->loadFilter('trimwhitespace');

// this will execute the template script,
// filter the output for white space,
// and then echo the filtered output.
$tpl->display('template.tpl.php');
?>
```

loadPlugin

```
void|Savant_Error loadPlugin ( string pluginName [, array options default null [,
boolean savantRef default false ] ] )
```

Loads a Savant plugin before it is called in a template.

If the pluginName cannot be found in the resource search path, returns a Savant_Error.

The options array is used to pass options into the plugin class.

If you set savantRef to true, a reference to the calling Savant object will be passed to the plugin object.

```php
<?php
$tpl =& new Savant2();

// pre-load the dateformat plugin with default options
$tpl->loadPlugin('dateformat');

// set up some custom options
$opts = array(
    'format' => '%Y-%m-%d' // the default date format
)

// preload the dateformat plugin with custom options
$tpl->loadPlugin('dateformat', $opts);


?>
```

loadTemplate

```
string|Savant _ Error loadTemplate ( [ string source default null [, boolean
setScript default false ] ] )
```

> *Security Note: As with all PHP functions that access the file system, you should be careful to check and/or sanitize any variables to you pass to this method, especially if they are accepted from an untrusted source (such as $_GET, $_POST, and $_REQUEST variables).*

Finds a template source from the template search path and, optionally, compiles the source into a PHP script. If the template source was successfully found (and optionally compiled) it returns the path to the template script; if not found or not successfully compiled, returns a Savant_Error object.

> *Note: By default, Savant does not compile template sources; the normal Savant template source is already a PHP script. Thus, the template source and the template script, by default, are the exact same file.*

If the source parameter is null, loadTemplate will find and optionally compiles the default template source from setTemplate().

You will rarely if ever need to pass the setScript parameter; this tells Savant to set the compiled script source as the "main" template script. The setScript parameter is used internally by Savant to track the original template script being executed.

The main use of loadTemplate() is to load templates from within other templates; for example, a header and/or footer template.

```
<html>

<!-- template source -->

<?php include $this->loadTemplate('header.tpl.php') ?>

<h2>Main Content</h2>
<p>Your name is <?php echo $this->name ?>, right?</p>

<?php include $this->loadTemplate('footer.tpl.php') ?>

</html>
```

plugin

```
void plugin ( string pluginName [, mixed option1 [, mixed option2 [, ...] ] ] )
```

Use this method inside your template source to call a plugin and output the results.

Because the template becomes part of Savant object when you call display() or fetch(), use the syntax $this->plugin() in your template source.

The plugin() method takes as many arguments as the plugin requires. For example, to call the stylesheet plugin (the parameter is the HREF to the stylesheet):

```
<html>
    <?php $this->plugin('stylesheet', 'path/to/styles.css'); ?>
</html>
```

... or to call the input plugin (the parameters are input-type, element-name, and element-value):

```
<html>
    <?php $this->plugin('input', 'text', 'name', $this->name); ?>
</html>
```

Savant will load and instantiate plugins with their default options as they are called in the template; if a plugin has been pre-loaded with loadPlugin(), the pre-loaded options will be used instead of the defaults.

setCompiler

```
void|Savant _ Error setCompiler ( &object compilerObject )
```

Sets a reference to the compiler/pre-processor object for Savant to use when loading template sources.

By default, Savant does not compile or pre-process templates; a regular Savant template is already in PHP, so it does not need to be compiled.

However, some applications may benefit from pre-processing or compiling; this allows the developer to "hook" a custom compiler object to Savant.

The compilerObject can be of any class, so long as it has a public compile() method (which should return a path or stream to a PHP script generated from the tempalte source).

If compilerObject is not an object, or does not have a public compile() method, setCompiler() will throw a Savant_Error and reset the internal compiler object to null (i.e., no compiler).

```php
<?php
require _ once 'My _ Compiler.class.php';
$compiler =& new My _ Compiler();

require _ once 'Savant2.php';
$tpl =& new Savant2();
$tpl->setCompiler($compiler);

// now when you call display() or fetch(), the template source
// will be passed through $compiler, which itself should return
// a path to the compiled PHP script for Savant2 to execute
?>
```

setError

```
void setError ( string errorClass default null )
```

Sets the name of the error class resource to use. If the errorClass is null, uses the default Savant_Error class.

```php
<?php
$tpl =& new Savant2();

// throw PEAR errors with the Savant _ Error _ pear class
$tpl->setError('pear');

// or use PEAR _ ErrorStack via the Savant _ Error _ stack class
$tpl->setError('stack');
?>
```

```
void setError ( string errorClass default null )
```

setExtract

```
void setExtract ( [ bool flag default false ] )
```

Turns variable extraction off and on with display() and fetch().

This method is provided as a service to users of Savant 1.x so that old templates can be ported forward to Savant2 more easily. New Savant2 template sources should use the $this->var notation.

When variable extraction is off (the default) you address assigned variables in the template source with $this->var notation.

```
<html>
    <h1><?php echo $this->title ?></h1>
</html>
With variable extraction turned on, you don't need to use $this.
<html>
    <h1><?php echo $title ?></h1>
</html>
```

The only reason to use setExtract() is to help in transferring your older Savant 1.x templates into Savant2. The convention is not to use setExtract() in Savant2 templates.

- While extracted variables are easier to read in some cases, it becomes difficult to distinguish between variables assigned to the template and variables created within the template.

- It is too easy for template-created variables to "step on" assigned variables.

- Extracting variables slows down Savant2 and uses up some more memory (for references to the assigned variable properties of the Savant2 object).

For example, in a foreach loop, with extraction turned off (the default):

```
<html>
    <?php foreach ($this->list as $key => $val): ?>
        <p>The key is <?php echo $key ?><br />
        and the value is <?php echo $val ?>.</p>
    <?php endforeach; ?>
</html>
```

The same loop with extraction turned on:

```
<html>
    <?php foreach ($list as $key => $val): ?>
        <p>The key is <?php echo $key ?><br />
        and the value is <?php echo $val ?>.</p>
    <?php endforeach; ?>
</html>
```

If the template had 'key' and 'val' assigned to it, the above loop just overwrote those values.

setPath

```
void setPath ( string type, string|array path )
```

Security Note: As with all PHP functions that access the file system, you should be careful to check and/or sanitize any variables to you pass to this method, especially if they are accepted from an untrusted source (such as $_GET, $_POST, and $_REQUEST variables).

Clears then sets the directories for a search path. The type parameter is always 'template' or 'resource', and the path parameter is either a path string or an array of directories.

For example:

```php
<?php
$tpl =& new Savant2();

// reset the template search path as a string
$tpl->setPath('template', '/path/one:/path/two');

// reset the resource search path as an array
$tpl->setPath('resource', array('/path/three', '/path/four'));

?>
```

setReference

```
void setReference ( bool pass Savant2 reference or not )
```

Plugin and filter objects have an optional, internal property called $Savant. This is used to hold a reference to the calling Savant2 object. By default, it is not set. Call "setReference(true)" to change the default behavior so that the internal $Savant property is populated with a reference every time a plugin or filter is loaded.

```
void setReference ( bool pass Savant2 reference or not )
```

setTemplate

```
void setTemplate ( string source )
```

> *Security Note: As with all PHP functions that access the file system, you should be careful to check and/or sanitize any variables to you pass to this method, especially if they are accepted from an untrusted source (such as $_GET, $_POST, and $_REQUEST variables).*

Sets the name of the default template source to use for display() and fetch() calls. Savant will use the template search path to find the template source.

```php
<?php
$tpl =& new Savant2();
$tpl->setTemplate('template.tpl.php');
?>
```

splugin

```
mixed splugin ( string pluginName [, mixed option1 [, mixed option2 [, ...] ] ] )
```

Use this method inside your template source to call a plugin and return (not output) the results, similar to sprintf.

Because the template becomes part of Savant object when you call display() or fetch(), use the syntax $this->splugin() in your template source.

The splugin() method takes as many arguments as the plugin requires. For example, to call the stylesheet plugin (the parameter is the HREF to the stylesheet):

```
<html>
    <?php $output = $this->splugin('stylesheet', 'path/to/styles.css'); ?>
</html>
```

... or to call the input plugin (the parameters are input-type, element-name, and element-value):

```
<html>
    <?php $output = $this->splugin('input', 'text', 'name', $this->name); ?>
</html>
```

Savant will load and instantiate plugins with their default options as they are called in the template; if a plugin has been pre-loaded with loadPlugin(), the pre-loaded options will be used instead of the defaults.

unloadFilter

```
void unloadFilter ( [ mixed filterName default null ] )
```

If a filter has been loaded, this will unset it so that it may be re-loaded with new options.

If the filterName is null, all filters will be unloaded. If filterName is a string, only that filter will be unloaded. If filterName is an array, only filters named as values in the array will be unloaded.

> *Note: If you unload a filter, it "loses its place" in the filter order. When you add it again, it will be at the end of the filter list.*

unloadPlugin

void **unloadPlugin** ([mixed pluginName default null])

If a plugin has been loaded, this will unset it so that it may be re-loaded with new options.

If the pluginName is null, all plugins will be unloaded. If pluginName is a string, only that plugin will be unloaded. If pluginName is an array, only plugins named as values in the array will be unloaded.