

The Savant Template System for PHP (Offline Edition)

The simple, elegant, and powerful alternative to Smarty.

API Reference

This version:

http://www.number0.info/public/doc/savant/savant_api.html

Official version:

<http://phpsavant.com/yawiki/index.php?area=Savant2&page=ApiRef>

Authors:

number0 Sebastian Siennicki.

www: <http://number0.info/>.

email: number0@number0.info

Last update:

3 / 08 / 2005

This document is also available in these non-normative formats: [PDF version](#) and [ZIP archive](#).

[Copyright](#) ©2005 [number0](#), [Savant](#).

Table of Contents

- General
 - [Savant2\(\)](#) instantiates a new Savant object
- Variables
 - [assign\(\)](#) assigns a variable by copy
 - [assignRef\(\)](#) assigns a variable by reference
 - [clear\(\)](#) unsets or clears an assigned variable
 - [getVars\(\)](#) gets the value of an assigned variable
- Escaping
 - [setEscape\(\)](#) sets the escaping callbacks
 - [addEscape\(\)](#) adds to the current escaping callbacks
 - [getEscape\(\)](#) returns the array of escaping callbacks
 - [escape\(\)](#) escapes and returns a value
 - [__\(\)](#) escapes and prints a value
- Templates
 - [setTemplate\(\)](#) sets the default template source
 - [display\(\)](#) executes then displays template script output
 - [fetch\(\)](#) executes then returns template script output
 - [loadTemplate\(\)](#) returns the path to a template script
- Path Management
 - [addPath\(\)](#) adds a template or resource directory to the current search path
 - [getPath\(\)](#) gets the current template or resource search path
 - [setPath\(\)](#) resets the template or resource search path
- Plugins

- [plugin\(\)](#) executes and displays plugin output
- [splugin\(\)](#) executes and returns plugin output
- [loadPlugin\(\)](#) loads a plugin class
- [unloadPlugin\(\)](#) unloads a plugin class
- Filters
 - [loadFilter\(\)](#) loads a filter to be applied to template output
 - [unloadFilter\(\)](#) unloads a filter so it is not applied to template output
- Error handling
 - [isError\(\)](#) checks if a variable is a Savant error
 - [setError\(\)](#) sets the name of the error resource class
- Compiler
 - [setCompiler\(\)](#) sets the compiler object

General

Savant2() - Constructor

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

This constructs a new Savant2 object.

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

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

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

Variables

assign()

Table of Contents

- [Assign by name](#)
- [Direct assignment](#)
- [Assign from associative array](#)
- [Assign from object properties](#)

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
$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>
  <title><?php $this->_($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
$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
$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
$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>

  <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 $this->_($val['author']) ?></td>
            <td><?php echo $this->_($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
$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
$tpl->assignRef('refvar', $referenced_variable);
?>
```

You can also assign references directly to the Savant object:

```
<?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
$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);
?>
```

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
$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
$tpl =& new Savant2();

$vars = get_object_vars($tpl);
?>
```

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

Escaping

setEscape()

```
public void setEscape ( )
```

Use `setEscape()` to clear out any previous escaping callbacks and replace them with an arbitrary number of new callbacks. Each parameter you pass to `setEscape()` is treated as an escaping callback. For example:

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

// set a single function callback
$tpl->setEscape('htmlspecialchars');

// set two function callbacks
$tpl->setEscape('stripslashes', 'htmlspecialchars');

// set a static-method callback
$tpl->setEscape(
    array('StaticClass', 'methodName')
);

// set an object-method callback
$tpl->setEscape(
    array($someObject, 'methodName')
);

// set all four, but in different combination
$tpl->setEscape(
    array($someObject, 'methodName'),
    'stripslashes',
    array('StaticClass', 'methodName'),
    'htmlspecialchars'
);
?>
```

Escaping callbacks are applied in order. The default callback is 'htmlspecialchars' until you set new callbacks.

The callback values you set must be suitable as a first parameter to [call_user_func\(\)](#). In addition, the function or method you specify should take the value to be escaped as the very first parameter, with no other required parameters, and should return an escaped copy of the value.

For example, this function would serve well as an escaping callback; note how it has only one required parameter, and returns the value after being escaped.

```
<?php
function my_escape($value)
```

```
{
    return str_replace('in-to', 'out-of', $value);
}
?>
```

addEscape()

```
public void addEscape ( )
```

Similar to [setEscape\(\)](#), this method appends callbacks to the current escaping callbacks (instead of clearing them out first). For example:

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

// set the escaping to just this one function
$tpl->setEscape('stripslashes');

// add another escaping callback to the current set
$tpl->addEscape('htmlspecialchars');

// now both stripslashes and htmlspecialchars
// will be applied when escaping values
?>
```

getEscape()

```
public array getEscape ( )
```

This method will return an array of the current escaping callbacks put in place by [setEscape\(\)](#) and [addEscape\(\)](#). For example:

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

// set up escaping callbacks
$tpl->setEscape(
    'stripslashes',
    array('StaticClass', 'methodName'),
    'htmlspecialchars'
);

// now see how they are stored in Savant
$result = $tpl->getEscape();
print_r($result);
?>
```

escape()

```
public mixed escape ( mixed value to be escaped )
```

Use this function to apply all escaping callbacks set with [setEscape\(\)](#) and [addEscape\(\)](#) to a value; the escaped version of the value is then returned. (The default escaping is 'htmlspecialchars'.)

For example, with this business logic ...

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

// set up escaping
$tpl->setEscape('stripslashes', 'htmlspecialchars');

// assign a value and display
$tpl->assign('value', "Here\'s one thing & another.");
$tpl->display(...);
?>
```

... and this template ...

```
<?php
echo $this->escape($this->value);

?>
```

... you would get this output:

```
Here's one thing & another.
```

Note that this method returns the escaped value, but does not print it by itself. To escape-and-print a value, use the [_\(\)](#) method.

[_\(\)](#)

```
public void _ ( mixed value to be escaped )
```

Use this function to apply all escaping callbacks set with [setEscape\(\)](#) and [addEscape\(\)](#) to a value; the escaped version of the value is then printed for you. (The default escaping is 'htmlspecialchars'.)

For example, with this business logic ...

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

// set up escaping
$tpl->setEscape('stripslashes', 'htmlspecialchars');

// assign a value and display
$tpl->assign('value', "Here\'s one thing & another.");
$tpl->display(...);
```

```
?>
```

... and this template ...

```
<?php
$this->_($this->value);
?>
```

... you would get this output:

```
Here's one thing & another.
```

Note that this method prints the escaped value automatically. To get an escaped copy of a value, use the [escape\(\)](#) method.

Templates

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
$tpl =& new Savant2();

$tpl->setTemplate('template.tpl.php');
?>
```

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
// 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
// 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();

?>
```

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>
```

Path Management

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
$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
$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
*/
?>
```

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.

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
$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'));

?>
```

Plugins

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.

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.

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
$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);

?>
```

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.

Filters

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
$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');

?>
```

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.

Error handling

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
$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;
}

?>
```

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
```

```
$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');
?>
```

Compiler

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 template 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
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
?>
```