

Zend_Html_Filter - Pádraic Brady

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

Zend Framework: Zend\Htm\Filter Component Proposal

Proposed Component Name	Zend\Htm\Filter
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend\Htm\Filter
Proposers	Pádraic Brady
Zend Liaison	TBD
Revision	1.0 - 10 August 2010: Initial Version. (wiki revision: 9)

Table of Contents

1. Overview
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
4. Dependencies on Other Framework Components
5. Theory of Operation
6. Milestones / Tasks
7. Class Index
8. Use Cases
9. Class Skeletons

1. Overview

Zend\Htm\Filter is a generic DOM filter for performing HTML sanitisation and other HTML manipulations. While its focus is obviously HTML sanitisation, it uses a filter queue approach that is also suitable for dynamically altering HTML via the PHP DOM with reusable filter objects. This generic approach allows for other uses such as replacing, altering or adding elements, attributes and custom styling to the HTML being filtered.

This component is being proposed to Zend Framework 2.0 to ensure that Zend Framework developers have access to HTML sanitisation features that do not rely on external libraries or require mixing previously established classes of limited scope such as `Zend_Filter_StripTags`, an approach that is often fundamentally flawed and unnecessarily complex. While there is nothing wrong per se with using external libraries, `Zend\Htm\Filter` was designed not simply as a handy Zend Framework component but as a full featured sanitiser which improves on existing external libraries' characteristics in both security and performance. The goal of `Zend\Htm\Filter` is to become the best performing fully featured HTML sanitiser in PHP. It will be made available outside of the Zend Framework (for non-framework users) as the `Wibble HTML Sanitiser`.

Having a HTML sanitiser in the Zend Framework would fill a large gap in its overall security environment. At present applications often end up consuming third-party HTML from sources such as RSS and Atom feeds, emails, web scraping, web APIs, blog comments, WYSIWYG editors, etc. It has always been essential that developers sanitise such input to ensure it does not introduce Cross-Site Scripting (XSS) and Phishing vulnerabilities, as well as ensuring that the resulting output meets the requirements of an acceptable HTML standard and is free from obvious attempts to break normal page layout.

At present, only `HTMLPurifier` is capable of all these tasks and Zend Framework developers have required custom filters and classes in order to integrate it for Zend Framework applications. All other alternatives to `HTMLPurifier` have been found to suffer from insecure behaviour, missing sanitisation coverage and a lack of dependable HTML tidying. These findings are based on my own examination of `HTMLPurifier` and its alternatives: [HTML Sanitisation: The Devil's In The Details \(And The Vulnerabilities\)](#). While this analysis may be biased in some respects (since I am a hardcore `HTMLPurifier` user), its factual findings are quite clear. Most HTML sanitisers are at present not all that secure.

`Zend\Htm\Filter` is intended to closely match `HTMLPurifier`'s capabilities but at a significantly reduced cost in terms of performance. This is

achieved by offloading complex HTML parsing and tidying to the PHP DOM and HTML Tidy extensions. It also improves performance by using DOM filters to strip, escape or prune HTML content considered dangerous. Performing this via DOM methods is often both faster and more reliable than doing so with regular expressions. This does not mean that `Zend\Html\Filter` will outperform common regular expression based HTML sanitisers in all scenarios. However, the more complex the task the more likely it is that regular expression processing will become a performance bottleneck, one that `Zend\Html\Filter` does not encounter, bringing their performance closer together. Rough benchmarks for the Wibble prototype that this proposal is based on can be found at: [HTML Sanitisation Benchmarking With Wibble \(ZF Proposal\)](#).

2. References

- [HTMLPurifier](#)
- [Zend\Html\Filter Prototype \(Wibble\)](#)
- [HTML Sanitisation Benchmarking With Wibble](#)

Please note that the available Wibble source code is a prototype and not a final implementation.

3. Component Requirements, Constraints, and Acceptance Criteria

- To perform full featured HTML Sanitisation
- To offer a generalised HTML filter mechanism to support decoration of output

4. Dependencies on Other Framework Components

This component has no dependencies on other framework classes.

5. Theory of Operation

`Zend\Html\Filter` operates as a DOM filter queue with a HTML Tidy postprocessor. In effect, it follows the following process:

1. Converts all HTML input to HTML safe UTF-8
2. Loads the HTML into a `DOMDocument` object
3. Applies one or more filters (DOM manipulators) to the HTML DOM
4. Extracts the filtered HTML from DOM and applies HTML Tidy (`ext/tidy`)
5. Converts the final HTML to the user's selected character encoding (if not UTF-8)

Many of these stages are self-explanatory, so the remainder of this section concern the filters.

The filters used by `Zend\Html\Filter` are basically classes which apply DOM operations (using the normal PHP 5 DOM API). This allows filters to add, remove or alter any part of the DOM. The reliance on DOM affords developers a common well understood API for manipulating HTML without resorting to regular expressions. Filters may be applied one after another as a series of filter operations (these are called explicitly and no internal queue is actually constructed).

By default, `Zend\Html\Filter` bundles filters named Strip, Escape, Prune and Cull for use in various settings for HTML sanitisation. The Strip filter is enabled by default and, surprise, strips out any HTML considered dangerous from an XSS or Phishing perspective. While `Zend\Html\Filter` includes a number of (provisional - to be reviewed) whitelists which may be used by these filters, the Strip filter assumes by default that all HTML elements must be stripped.

All of the bundled filters also make use of a set of utility methods dedicated to applying any whitelists and sanitisation to key parts of the HTML contained in a `DOMDocument`. Whitelisting prevents the occurrence of elements or attributes which are either illegal or not safe for output. The sanitisation prevents the occurrence of attribute values and CSS values which are likewise illegal or unsafe. This part of `Zend\Html\Filter` does utilise regular expressions though they are not subject to being fooled by malformed string input (as is the major risk with purely regular expression driven sanitisers). `Zend\Html\Filter` not only normalises all input to UTF-8, it also then removes UTF-8 characters which are recommended to be avoided (for example, it would remove half-width characters which are interpreted by IE6 and capable of use as XSS vectors). The regular expressions are "borrowed" from several open source HTML sanitisation and parsing libraries from outside PHP. This can be perceived as an odd decision, however it is foolhardy to reinvent the wheel for a handful of one line regular expressions already widely deployed and tested in multiple other languages outside of PHP. All sanitisation routines operate on a zero tolerance basis by removing failing attribute or CSS values instead of attempting to manipulate them into compliance.

Finally, once all sanitisation/filtering has been performed, the `DOMDocument` output is passed through `ext/tidy`. This postprocessing stage ensures that the output HTML is well formed and complies to a user selected HTML standard. While this stage is absolutely necessary to guarantee well formed output, it may be disabled via an option (an exception is thrown by default on systems not supporting `ext/tidy`).

Zend\Htm\Filter was specifically tested to ensure that disabling HTML Tidy did not impact on the efficacy of its sanitisation (many sanitisation related unit tests were performed under both conditions) of XSS and Phishing vulnerabilities though it may allow other page breaking tactics (hence the need to explicitly choose to disable HTML Tidy).

Finally, it is important to note that Zend\Htm\Filter cannot help but not support the entire scope of HTML 5. Neither libxml2 (the basis of PHP DOM) or HTML Tidy understand HTML 5 at the time of this proposal. There are HTML5 elements and other valid syntax which would be stripped from HTML 5 input. This does not mean that you can't filter a typical HTML 5 snippet and include it into HTML 5 - the limitations largely surround new HTML elements introduced with HTML 5 such as, for example, the section and time elements.

6. Milestones / Tasks

- Milestone 1: Complete Proposal Process for acceptance
- Milestone 2: Migrate Wibble prototype to final design (post discussion with CR Team)
- Milestone 3: Complete security review procedures (various peer reviews and testing)
- Milestone 4: Complete additional tasks such as documentation and writing some commonly useful filters

7. Class Index

- Zend\Htm\Filter\AbstractFilter
- Zend\Htm\Filter\Filterable
- Zend\Htm\Filter
- Zend\Htm\Filter\Strip
- Zend\Htm\Filter\Cull
- Zend\Htm\Filter\Escape
- Zend\Htm\Filter\Prune
- Zend\Htm\Filter\Utility
- Zend\Htm\Filter\Html\Document
- Zend\Htm\Filter\Html\Fragment

8. Use Cases

UC-01: Strip all HTML as is the default and thus shortest use case

```
$filter = new Zend\Htm\Filter('This is <strong>Sparta</strong>!');  
echo $filter->filter(); // This is Sparta!
```

The behaviour of Zend\Htm\Filter is a bit different than what may be expected from a filter. Since it effectively encapsulates the HTML, a new Zend\Htm\Filter object is required for each HTML sample being filtered. This is similar to instantiating a DOMDocument for each new HTML sample.

The default filter is Zend\Htm\Filter\Strip. This is a whitelist based HTML sanitiser that sanitises HTML elements, attributes and attribute values, and also all CSS properties and values.

UC-02: Filter HTML using a named filter and a user whitelist

```
$filter = new Zend\Htm\Filter('This is <strong>Sparta</strong>!');  
echo $filter->filter('strip', array(  
    'strong' => array()  
)); // This is <strong>Sparta</strong>!
```

Whitelists may be passed to the filter() method as an array where each key is the element name, and each sub-array the attributes allowed for that element. Where no attributes are allowed, an empty array must be included. It's expected that Zend\Htm\Filter will adopt an alternative syntax for whitelists which conforms to the current syntax for HTMLPurifier (this would be far shorter and intuitive).

UC-03: Filter HTML using an escape filter

```
$filter = new Zend\Htm\Filter('This is <invalid><em>Sparta</em></invalid>!');  
echo $filter->filter('escape'); // This is &lt;invalid&gt;<em>Sparta</em>&lt;/invalid&gt;!
```

The Escape, Cull and Prune filters are a bit different to the Strip filter. The Strip filter is the HTML Sanitisation function for this component and is subject to a restrictive whitelist. The remaining filters also sanitise HTML, but their whitelists are based on valid HTML markup considered safe for output. The Cull filter is actually identical to the Strip filter except it uses the broader internal whitelist. The Escape filter operates on the same basis but escapes unsafe/illegal tags rather than removing them. The Prune filter removes unsafe/illegal tags (as per the internal whitelist) including all of their content and child elements.

UC-04: Filter HTML using multiple filters

```
$filter = new Zend\Html\Filter('This is <strong style="color:red;">Sparta</strong>!');
echo $filter->filter('strip', array('strong'=>array()))->filter('capitalizeStrongTerms'); //This is
<strong>SPARTA</strong>!
```

Filters may be chained together if required. The second filter above does not actually exist but could be added by a user (see below).

UC-05: Filter HTML using a closure

```
$filter = new Zend\Html\Filter('This is <strong>Sparta</strong>!');
echo $filter->filter($func = function($node) use ($func) {
    if ($node->nodeType == XML_ELEMENT_NODE && $node->tagName == 'strong') {
        $dom = $node->ownerDocument;
        $em = $dom->createElement('em');
        $node->parentNode->insertBefore($em, $node);
        $children = $node->childNodes;
        if ($children != null && $children->length > 0) {
            foreach ($children as $child) {
                $em->appendChild($child);
                $func($child);
            }
        }
        $node->parentNode->removeChild($node);
    }
});
// This is <em>Sparta</em>!
```

Instead of writing new filter classes, one may use closures for quick or once off uses.

UC-06: Filter HTML using a custom filter class

```
$filter = new Zend\Html\Filter('This is <strong>Sparta</strong>!');
echo $filter->filter(new StrongToEm); // This is <em>Sparta</em>!

class StrongToEm extends \Zend\Html\Filter\AbstractFilter
{
    public function filter(\DOMNode $node)
    {
        if ($node->nodeType == XML_ELEMENT_NODE && $node->tagName == 'strong') {
            $dom = $node->ownerDocument;
            $em = $dom->createElement('em');
            $node->parentNode->insertBefore($em, $node);
            $children = $node->childNodes;
            if ($children != null && $children->length > 0) {
                foreach ($children as $child) {
                    $em->appendChild($child);
                    $this->filter($child);
                }
            }
            $node->parentNode->removeChild($node);
        }
    }
}
```

Filters may extend `Zend\Html\Filter\AbstractFilter` or implement `Zend\Html\Filter\Filterable`.

9. Class Skeletons

No class skeletons are available. A prototype is included in the references for this proposal but does not necessarily reflect a final implementation.

```
]]></ac:plain-text-body></ac:macro>  
]]></ac:plain-text-body></ac:macro>
```