

This is **G o o g l e**'s cache of <http://framework.zend.com/manual/pt-br/zend.filter.input.html> as retrieved on Mar 12, 2007 17:06:12 GMT.

G o o g l e's cache is the snapshot that we took of the page as we crawled the web.

The page may have changed since that time. [Click here for the current page without highlighting.](#)

This cached page may reference images which are no longer available. [Click here for the cached text only.](#)

To link to or bookmark this page, use the following url: [http://www.google.com/search?](http://www.google.com/search?q=cache:cU0Ti_gd2y4J:framework.zend.com/manual/en/zend.filter.input.html+zend_filter_input&hl=en&ct=clnk&cd=2&gl=us)

[q=cache:cU0Ti_gd2y4J:framework.zend.com/manual/en/zend.filter.input.html+zend_filter_input&hl=en&ct=clnk&cd=2&gl=us](http://www.google.com/search?q=cache:cU0Ti_gd2y4J:framework.zend.com/manual/en/zend.filter.input.html+zend_filter_input&hl=en&ct=clnk&cd=2&gl=us)

Google is neither affiliated with the authors of this page nor responsible for its content.

These search terms have been highlighted: **zend_filter_input**

11.4. Zend_Filter_Input

11.4.1. Introduction

Zend_Filter_Input provides simple facilities that promote a structured and rigid approach to input filtering. Its purpose is multifaceted, because it caters to the needs of three different groups of people:

Developers

Although filtering input can never be as easy as doing nothing, developers need to ensure the integrity of their data without adding unnecessary complexity to their code. **Zend_Filter_Input** offers simple methods for the most common use cases, extensibility for edge cases, and a strict naming convention that promotes code clarity.

Managers

Managers of all types who need to maintain control over a large group of developers can enforce a structured approach to input filtering by restricting or eliminating access to raw input.

Auditors

Those who audit an application's code need to quickly and reliably identify when and where raw input is used by a developer. The characteristics that promote code clarity also aid auditors by providing a clear distinction among the different approaches to input filtering.

There are a variety of approaches to input filtering, and there are also a variety of facilities that PHP developers can use. Whitelist filtering, blacklist filtering, regular expressions, conditional statements, and native PHP functions are just a few examples of the input filtering potpourri. **Zend_Filter_Input** combines all of these facilities into a single API with consistent behavior and strict naming conventions. All of the methods abide by a simple rule - if the data is valid, it is returned, otherwise FALSE is returned. Extreme simplicity.

11.4.1.1. Whitelist Filtering

Whitelist filtering methods begin with the `is` prefix, such as `isAlpha()` and `isEmail()`. These methods inspect input according to pre-defined criteria and return the data only if it adheres to the criteria. If not, FALSE is returned. The following provides a simple demonstration:

```
<?php
$filterPost = new Zend_Filter_Input($_POST);

if ($alphaName = $filterPost->isAlpha('name')) {
    /* $alphaName contains only alphabetic characters. */
} else {
    /* $alphaName evaluates to FALSE. */
}
```

```
}
?>
```

This approach errs on the side of caution by performing a boolean evaluation of the return value. If you want to distinguish among values that evaluate to FALSE in PHP (such as the integer 0 and the empty string), you can perform a strict comparison to FALSE:

```
<?php
$filterPost = new Zend_Filter_Input($_POST);
$alphaName = $filterPost->isAlpha('name');

if ($alphaName !== FALSE) {
    /* $alphaName contains only alphabetic characters. */
} else {
    /* $alphaName === FALSE */
}

?>
```

11.4.1.2. Blind Filtering

Blind filtering methods begin with the get prefix, such as getAlpha() and getDigits(). These methods do not inspect input but instead return the portion of it considered to be valid. For example, getAlpha() returns only the alphabetic characters, if there are any. (If not, the remaining string is the empty string.) The following provides a simple demonstration:

```
<?php
/* $_POST['username'] = 'John123Doe'; */

$filterPost = new Zend_Filter_Input($_POST);
$alphaUsername = $filterPost->getAlpha('username');

/* $alphaUsername = 'JohnDoe'; */

?>
```

11.4.1.3. Blacklist Filtering

Blacklist filtering methods begin with the no prefix, such as noTags() and noPath(). These methods are identical to the blind filtering methods, except the criteria they enforce is based upon what is considered invalid instead of what is considered valid. Invalid data is removed, and the remaining data (assumed valid) is returned. The following provides a simple demonstration:

```
<?php
/* $_POST['comment'] = '<b>I love PHP!</b>'; */

$filterPost = new Zend_Filter_Input($_POST);
$taglessComment = $filterPost->noTags('comment');

/* $taglessComment = 'I love PHP!'; */

?>
```

11.4.2. Theory of Operation

Zend_Filter_Input consolidates a few distinct approaches to input filtering into a single API with consistent behavior and strict naming conventions (see Seção 11.4.1, "Introduction"). These characteristics bring **Zend_Filter_Input** on par with existing solutions, but they do nothing to further aid those who require a more structured or rigid approach. Therefore, by default, **Zend_Filter_Input** enforces controlled access to input.

Two syntaxes are supported. In the default (strict) approach, a single argument is passed to the constructor - the array to be filtered:

```
<?php
    $filterPost = new Zend_Filter_Input($_POST);
    $email = $filterPost->isEmail('email');
?>
```

Zend_Filter_Input sets the array that is passed (`$_POST`) to `NULL`, so direct access is no longer possible. (The raw data is only available through the `getRaw()` method, which is much easier to monitor and/or avoid altogether.)

In the optional (non-strict) approach, `FALSE` is passed as the second argument to the constructor:

```
<?php
    $filterPost = new Zend_Filter_Input($_POST, FALSE);
    $email = $filterPost->isEmail('email');
?>
```

The use of the filter is exactly the same, but **Zend_Filter_Input** does not set the original array (`$_POST`) to `NULL`, so developers can still access it directly. This approach is discouraged in favor of the strict approach.

Zend_Filter_Input is designed primarily with arrays in mind. Many sources of input are already covered by PHP's superglobal arrays (`$_GET`, `$_POST`, `$_COOKIE`, etc.), and arrays are a common construct used to store input from other sources. If you need to filter a scalar, see Capítulo 11, *Zend_Filter*.

11.4.3. Use Cases

Strict Whitelist Filtering (Preferred):

```
<?php
    $filterPost = new Zend_Filter_Input($_POST);
    if ($email = $filterPost->isEmail('email')) {
        /* $email is a valid email format. */
    } else {
        /* $email is not a valid email format. */
    }
?>
```

Strict Blind Filtering:

```
<?php
    $filterPost = new Zend_Filter_Input($_POST);
    $alphaName = $filterPost->getAlpha('name');

?>
```

Strict Blacklist Filtering:

```
<?php
    $filterPost = new Zend_Filter_Input($_POST);
    $taglessComment = $filterPost->noTags('comment');

?>
```

Non-Strict Whitelist Filtering:

```
<?php
    $filterPost = new Zend_Filter_Input($_POST, FALSE);
    if ($email = $filterPost->isEmail('email')) {
        /* $email is a valid email format. */
    } else {
        /* $email is not a valid email format. */
    }

?>
```

Non-Strict Blind Filtering:

```
<?php
    $filterPost = new Zend_Filter_Input($_POST, FALSE);
    $name = $filterPost->getAlpha('name');

?>
```

Non-Strict Blacklist Filtering:

```
<?php
    $filterPost = new Zend_Filter_Input($_POST, FALSE);
    $comment = $filterPost->noTags('comment');

?>
```