Zend_Filter Design Proposal - Darby Felton

<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_Filter Design Component Proposal

Proposed Component Name	Zend_Filter Design
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend_Filter Design
Proposers	Darby Felton, author & Zend liaison
Revision	0.3.0 - 10 Jan 2007: Resolved problems with dual-purpose component design and class naming schema (wiki revision: 22)

Table of Contents

1. Overview

Introduction

What is a filter?

In the real world, a filter is typically used for removing unwanted portions of input, and the desired portion of the input passes through as filter output (e.g., coffee). In such scenarios, a filter is an operator that produces a subset of the input. This type of filtering is useful for web applications - removing illegal input, trimming unnecessary white space, etc.

The idea of filtering may also be extended to performing generalized transformations upon input, more than simply acting as an operator to produce a selected subset of the input. A common transformation applied in web applications is the escaping of HTML entities. For example, if a form field should have its value populated with a dynamic value, this value should either be free of HTML entities or contain only escaped HTML entities, in order to prevent undesired behavior and security vulnerabilities. To meet this requirement we can either remove HTML entities or escape them, and other approaches may more appropriate for different situations. A filter that removes the HTML entities operates within the scope of the first definition of filter - an operator that produces a subset of the input. A filter that escapes the HTML entities, however, transforms the input. If we consider such use cases within the scope of filtering, then filtering must be redefined as an operator that performs transformations upon input.

What is a validator?

A validator examines its input with respect to some requirements and produces a boolean result - whether the input successfully validates against the requirements. If the input does not meet the requirements, a validator may additionally provide information about which requirement(s) the input does not meet.

For example, a web application might require that a username be between six and twelve characters in length and may only contain alphanumeric characters. A validator can be used for ensuring that usernames meet these requirements. If a chosen username does not meet one or both of the requirements, it would be useful to know which of the requirements the username fails to meet.

A validator differs from a filter primarily in that a validator produces a boolean result, based on whether the input validates against some requirements. A filter, on the other hand, produces output that is based on transformations upon the input.

Why consider filters and validators together?

Though filters and validators perform different operations upon their input, it is easy to see how filters and validators are often used together in web applications.

Consider the use case of a blog application that accepts comments upon its entries. Let us assume that the comment form contains two fields - an e-mail address field and a multi-line comment text area.

Let us further assume that a submitted comment containing an e-mail address that does not conform to the e-mail address schema should be rejected (e.g., not@realaddress). A validator would be used to determine whether an e-mail address is acceptable.

Since an accepted comment would appear on the associated article's web page, comment authors should be unable to perform cross-site scripting (XSS). A filter would be used to remove XSS-capable content or transform it such that XSS cannot be performed.

Because the e-mail validator and comment filter would be operating upon different parts of the same form input, they should be readily available to the same parts of the web application code. Thus, filters and validators are often used together in applications.

Filters and validators may also be related further than the proximity of their use in web application code. Consider a user-entered phone number as [part of] the application input. The phone number (555) 555-1234 may be written in several ways, such as 555.555.1234 and 555-555-1234, all of which should be considered valid. Though there are other ways to solve this problem, let us assume that the application will strip all punctuation and white space from the input, normalizing the phone number to contain only numbers. Once the normalized phone number is available from the filter, the application requires that the phone number be ten digits, and a simple validator can be used for meeting this requirement. Thus, for a single datum, the input phone number, the application must use a filter and then a validator. The filter and validator for a phone number should be readily and equally available to the application, perhaps even contained in the same class.

Zend_Filter already provides filters and validators. Why revisit these topics now?

The Zend_Filter component currently provides a library of static filter and validator methods via a monolithic `Zend_Filter` class.

A method of `Zend_Filter` is a filter or validator by naming convention only, and filter methods are further divided into whitelist and blacklist filters, again only by naming convention.

The Zend_Filter component includes another class, `Zend_Filter_Input`, that allows instantiation with an array of input data. The methods of `Zend_Filter_Input` generally act as proxies to the static methods of `Zend_Filter`, passing along a specific element of the input array to the proxied methods.

By using Zend_Filter an application gains only the filters and validators provided by the class, nothing more. The size of Zend_Filter increases proportional to the number of filters and validators. A class having too many methods is confusing and difficult to use, and the current `Zend_Filter` class cannot mitigate this risk except through potentially complicated naming conventions, which still does not reduce the number of methods in the class.

The only way for users to add functionality to Zend_Filter is by extending the `Zend_Filter` and `Zend_Filter_Input` classes. By doing so, a user class inherits all the filters and validators of each inherited class, even if only one or two methods are needed. This approach also does not support clean integration of filters and validators from third-party providers.

Complex filters and validators that require supporting class constants, instance variables, and/or helper methods are all contained in a single class. As the number of filters and validators grows, the complexity of Zend_Filter and its naming conventions grows proportionally.

Zend_Filter supports ordering of filters and validators only by the order in which the methods are called from user code. Though this may be enough, it may also be worthwhile to consider providing additional support for managing filters and validators. For example, if an ordered set of filters and/or validators are to be applied to several input data, it may be useful to have a container for the filters and/or validators that can operate on multiple input data and provide aggregated results to the user. `Zend_Filter_Input` provides some degree of support for this idea, but its limitations may hinder users.

In order to solve these limitations of the current Zend_Filter design, we revisit these topics in order to improve the component's design with respect to two key framework goals - extremely simple (to use) and use-at-will architecture (flexibility and extensibility).

Summary

Filters and validators are related but different operators. A filter performs some transformations on its input, and a validator returns a boolean result - whether its input meets certain requirements. They are often used at the same time in application code, and it is important to have filters and validators organized in such a way as to be readily available to various parts of the application, such as processing form data and escaping data for various media (e.g., HTML, RDBMS).

Special Thanks

Thanks to everyone who has helped with Zend_Filter to date, and the following people are recognized for their extraordinarily valuable contributions along the way:

- Andi Gutmans
- Chris Shiflett
- Christopher Thompson
- Gaylord Aulke
- Kevin McArthur
- Marco Tabini

Forgot about you? So sorry; please remind me!

2. References

- Zend_FilterChain Zend_Validator Component Proposal
- Intercepting Filters Component Proposal
- Symfony
    - Filters - symfony advent calendar day eighteen
    - How to validate a form
- PHP Filter Extension

3. Component Requirements, Constraints, and Acceptance Criteria

- The component must be simple to use.
- The component must be extensible. Users must be able to easily write and use filters and validators for their own purposes and publish those filters and validators for others to plug into their framework-powered applications.
- Filters and validators must be readily available throughout the lifetime of a request to the application. Input filtering and validation, for example, may be used early in the controller execution, whereas escaping output should be done as late as possible, closer to the view logic.
- Filters and validators are generally considered for inclusion with the framework component where the filter or validator helps the component meet the 80/20 rule with respect to solving common use cases.
- Filter and validator configuration and setup must be supported by an object-oriented syntax (e.g., using fluent interfaces).
- Filters and validators as a general rule do not throw exceptions. Filters and validators may throw exceptions, however, only when filtering or validation is reasonably impossible. Validators return true or false, depending on whether the input meets the validation criteria, and, in the event that the input fails validation, messages are accumulated and retained for programmatic access by the consuming application.
- Ordered filter and validator chaining must be supported.
- Validator chains must support validation rule dependencies, aborting execution when certain validators return false. Put another way, if there are two validators in a chain, and the second validator need not run if the input fails the first validation, then the chain must allow the developer to terminate the chain execution if the first validator returns false.
- PHP editor code completion should be supported as much as possible under the proposed design.
- Overloading with magic functions (e.g., `__get()`, `__call()`) should be avoided where possible, and all such usage must be documented well for ease of use.

4. Dependencies on Other Framework Components

- Zend_Exception

5. Theory of Operation

Organizing Filters and Validators into Classes

Monolithic Class

In this approach, all filters and validators are lumped into a single, monolithic class, whereby the class essentially acts as a library of filtering and validation functions. The current Zend_Filter component in the framework core realizes this approach.

Category Classes

This approach divides the class of the monolithic approach into classes that group filters and validators by purpose. Filters and validators may be categorized into multiple classes based on their general purpose. For example, we might have a String class that provides dozens of filtering and validation methods within its scope. A Number class could be used to provide methods for filtering and validating numeric input. As another example, we could have a Location class that is used to provide methods for dealing with address or location information (e.g., city, country, postal code, latitude and longitude).

Atomic Classes

This approach reduces the class size to contain one filter, one validator, or both. A class that needs to filter would implement a documented filter interface. If a class needs to validate, it would implement a documented validator interface. Because PHP allows for a class to implement multiple interfaces, a class could be both a filter and a validator. Some examples might include IntegerRange, RegExp, and CreditCard.

Choice Comparisons

Criterion	Monolithic Class	Category Classes	Atomic Classes
Filter implements a documented interface	resort to documented conventions	resort to documented conventions; possibly implement an interface	implement an interface
Validator implements a documented interface	resort to documented conventions	resort to documented conventions; possibly implement an interface	implement an interface
Extensibility	user classes inherit unnecessary complexity	user classes must conform to well-documented conventions, rather than conforming to simple, well-documented interfaces; higher complexity than Atomic approach	user classes implement simple, well-documented interfaces
Class Complexity	high	medium	low
Number of Classes	low	medium	high
Naming Conventions Complexity		added complexity of having to categorize among various classes	organization of classes needed
Ease of Use		conventions add some complexity	organization of classes needed
Use-at-will Architecture	unavailable	category classes can be loaded at will, though some may contain unnecessary weight	all filters and validators can be loaded at will
Chaining Filters & Validators	requires conventions	requires conventions; added complexity of categories	chaining is simple because objects implement known interfaces

Summary

As the above comparison table indicates, the monolithic class approach has the most trouble adequately meeting the component criteria.

The category class approach improves upon having a monolithic class, though the categorization of the filters and validators introduces its own complexity into the design, particularly because some categories are likely to overlap (i.e., not being mutually exclusive) with other categories. The names of categories are likely to be interpreted in different ways by the user community, perhaps causing additional confusion.

Of the three approaches, the atomic class approach seems to meet the criteria the best, with one caveat. Because the number of classes increases proportionally to the number of filters and validators, and because we need to maintain a reasonable number of classes in each directory, it may be necessary to group or categorize the classes using the normal PEAR conventions already in use throughout the framework. This implies that some additional complexity due to such grouping would be necessary, but this is not the same complexity that is introduced in the category class approach, wherein each class contains many filters and validators that correspond to a single category.

6. Milestones / Tasks

1. Publish design notes - **DONE**
2. Publish proposal based on design notes - **DONE**
3. Arrive on proposal approvable for incubator development
4. Commit working prototype to incubator
5. Commit passing unit tests
6. Write initial documentation
7. Revise code, tests, and docs based on feedback
8. Merge changes with trunk for core release

7. Class Index

- Zend_Filter
- Zend_Filter_Exception
- Zend_Filter_Interface
- Zend_Filter_HtmlEntities
- Zend_Filter_StringToLower
- Zend_Filter_StringTrim
- Zend_Filter_StripTags
- Zend_Validate_Exception
- Zend_Validate_Interface
- Zend_Validate_EmailAddress
- Zend_Validate_StringLength
- (additional classes from porting existing Zend_Filter methods omitted for brevity)

8. Use Cases

9. Class Skeletons

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