

# Refactoring Zend.php

<ac:macro ac:name="note"><ac:parameter ac:name="title">Under Construction</ac:parameter><ac:rich-text-body>  
<p>Please edit this page and add your ideas about how to "fix" Zend.php. Ideally, the resulting architecture would support extensibility without sacrificing the hallmark simplicity of Zend Framework.</p></ac:rich-text-body></ac:macro>

<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: What to do with Zend.php Component Proposal

<b>Proposed Component Name</b>	What to do with Zend.php
<b>Developer Notes</b>	<a href="http://framework.zend.com/wiki/display/ZFDEV/What+to+do+with+Zend.php">http://framework.zend.com/wiki/display/ZFDEV/What to do with Zend.php</a>
<b>Proposers</b>	Zend Framework community
<b>Revision</b>	0.8 - 26 February 2007: Created from community discussion on fw-general and ZF-958. 0.8.1 - 28 February 2007: Trying to outline various possible solutions 0.8.2 - 6 March 2007: Add option E (wiki revision: 33)

## Table of Contents

1. Overview
- Solution A:
- Solution B:
- Solution C:
- Solution D:
- Solution E:
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
- @TODO
4. Dependencies on Other Framework Components
5. Theory of Operation
6. Milestones / Tasks
7. Class Index
8. Use Cases
9. Class Skeletons

## 1. Overview

There has been a long discussion going on about the `Zend.php` file, and what should happen with it. Apparently, [TIMTOWTDI!](#)

Need to distill the following into explicit requirements .. feel free to volunteer 😊

Splitting up `Zend.php` does not necessarily require creating a bunch of tiny classes.

However, consider proposals like [Zend Core include path utilities - Aleksey V. Zapparov](#), where some developers might like to "include" some useful utility functions from their application bootstrap. Ideally, the ZF would include a mechanism and architecture to facilitate this, without just appending more and more functions to `Zend.php`

We'll try to put all of the suggestion in line, as it is hard to keep on overview.

## Some example justifications for splitting / cleaning up Zend.php

- Why is Zend.php called Zend.php? Cleaning up the organization via splitting also helps reduce naming confusion amongst beginners.
- Solves the svn externals issue described in detail previously.
- Why should I load a "debug" function with every request?
- How can I add more debug functions for use when I do want/need it (possibly loaded on-demand, even in my production environment) in a manner consistent with existing debug functions that is "compatible" with other related functions other developers might contribute (i.e. where is the standardization to facilitate community development and enhancement of framework-level "bootstrap" type features?)
- Where is a standard mechanism for developers to add "global" functionality to their ZF app (e.g. "plugins" for Zend.php)
- Why is the registry accessed via the oddly named functions in Zend.php? (Yes, BC, but it would be nice to switch to something cleaner.)
- How should developers extend the functionality mixed into the current Zend.php? Need to plan for the long-term here ...

---

## Solution A:

Move Zend.php to <zf-home>/library/Zend/Zend.php. The class name would continue to be Zend, and no methods within the class will be changed.

The only change in your application would be that you need to change this:

```
require_once 'Zend.php';
```

To this:

```
require_once 'Zend/Zend.php';
```

Alternatively, you could add the Zend directory to your include\_path.

### The benefits

- Zend Framework files contained in a single directory. This simplifies integration of a Zend Framework tree into an application tree. You could use svn:externals, for instance.
- No usage of this class would change. Thus, no API change in the 0.9 release.

### The disadvantages

- The disadvantage is that it does not follow the ZF convention that class names always match the physical location of the class file. But this is the only class that needs to be an exception to this convention.
- Continued use of "bucket class" does not necessarily address future functionality needs

---

## Solution B:

Rename Zend.php to Zend/Core.php or Zend/Utility.php and leave it as a general "core" or "utility" class. Of course, we could mark the original library/Zend.php as {@deprecated}, having class Zend temporarily proxy to whatever classes we decide for delegation.

### The benefits

- Zend Framework files contained in a single directory. This simplifies integration of a Zend Framework tree into an application tree. You could use svn:externals, for instance.

### The disadvantages

- Continued use of "bucket class" does not necessarily address future functionality needs
- 

## Solution C:

```
static public function loadClass($class, $dirs = null)
static public function loadFile($filename, $dirs = null, $once = false)
static public function isReadable($filename)
```

Will move to a basic class like `Zend/Loader.php`, or a more full-fledged class like `Zend/FileSystem.php`, which could also include recursive directory iterators, regular expressions-based file manipulation, etc.

```
static public function loadInterface($class, $dirs = null)
static public function exception($class, $message = '', $code = 0)
```

Will be deprecated (removed).

```
static public function register($index, $newval)
static public function registry($index = null)
static public function isRegistered($index)
static public function initRegistry($registry = 'Zend_Registry')
static public function __unsetRegistry()
```

Will be superseded by `Zend/Registry.php`.

```
static public function dump($var, $label=null, $echo=true)
```

Will move to `Zend/Debug.php`, which could include a whole host of additional debugging information, interaction with `Zend_Log` for trace messages, pseudo-breakpoints, etc.

And last but not least:

```
static public function compareVersion($version)
```

Will either be handled by `Zend/Environment.php` or move to something like `Zend/Version.php`

### The benefits

- Zend Framework files contained in a single directory. This simplifies integration of a Zend Framework tree into an application tree. You could use `svn:externals`, for instance.
- Modular; you don't have to load it if you don't use it
- Consistent; no confusing exceptions or non-standard practices
- Easily add additional functionality at a future date without bloating a core class (additional file loading or debugging methods, for example)
- Eliminates `Zend.php` class altogether---class is sometimes seen as confusing for new users who think they are including the entire framework or don't understand the purpose of the class

### The disadvantages

- May break backwards compatibility for applications built before 0.9 (beta). (However, it is better to break compatibility now rather than sometime after 1.0.)
- May need to add one or two additional `require_once` statements to bootstrap file

---

## Solution D:

We still believe having all the core functionality in one file is beneficial. While we want to keep the framework as loosely coupled as possible there are some key APIs which are pretty much used universally throughout the framework. Breaking them each out into separate classes and files would probably lead to a real performance hit and would require people to `require_once()` all those components over and over again (which would also be a PITA).

We therefore suggest that these key classes (previously known as `Zend:::`) all are put in the `Zend/Core.php` file.

```
library/Zend/Core.php:
-----

class Zend_Registry {
    static public function register($index, $newval) null);
    static public function registry($index = null);
    static public function isRegistered($index);
    static public function initRegistry($registry = 'Zend_Registry');
    static public function __unsetRegistry();
}

class Zend_Loader {
    static public function loadClass($class, $dirs = null);
    static public function loadFile($filename, $dirs = null, $once =
false);
    static public function isReadable($filename);
}

class Zend_Framework {
    static public function compareVersion($version)
    const VERSION = '0.8.0dev';
}

class Zend_Exception { ... }

// Could live with the following outside of Core.php but it's so small
it really isn't a big deal to keep it for convenience
class Zend_Debug {
    static public function dump($var, $label=null, $echo=true)
}
}
```

### The benefits

- Zend Framework files contained in a single directory. This simplifies integration of a Zend Framework tree into an application tree. You could use `svn:externals`, for instance.
- Simplifies inclusion into applications, with four fewer `require_once` statements

### The disadvantages

- Multiple classes in one file; non-standard practice
- May have to load classes that are not ever used (for example, `Zend_Registry`)
- Might be premature optimization

---

## Solution E:

Very similar to Solution C, but with minor variation on names of classes.

- Methods for class-loading move to `Zend/Loader.php`.
- Method `isReadable()` also moves to `Zend/Loader.php` because it doesn't justify having another class for this one method. If we develop a `Zend/Filesystem.php` class in the future, we'll move `isReadable()` and deprecate it in `Zend/Loader.php`.
- Redesign Registry methods to store a static object in the `Zend/Registry.php` class. Allow developer to specify which class to use for this static instance, to allow subclassing. But of course you can also create an instance of `Zend_Registry` in a traditional manner, with `new`.
- Registry also has static methods `get()` and `put()`, which correspond to the old `Zend::registry()` and `Zend::register()` methods. Internally, these methods are simply `$r = self::getInstance(); return $r[key];` and `$r = self::getInstance(); $r[key] = value;`.
- Method `and const` for version moves to `Zend/Version.php`.
- Method `dump()` moves to class `Zend_Debug`.
- Rewrite `Zend.php` methods to proxy to new classes, and mark them deprecated in ZF 0.9.0. Remove `Zend.php` in the subsequent ZF 1.0.0 RC1.
- Rewrite unit tests.
- Rewrite manual pages.

### The benefits

- Zend Framework files contained in a single directory. This simplifies integration of a Zend Framework tree into an application tree. You could use `svn:externals`, for instance.
- Modular; you don't have to load it if you don't use it.
- Consistent; no confusing exceptions or non-standard practices
- Easily add additional functionality at a future date without bloating a core class (additional file loading or debugging methods, for example)
- Eliminates `Zend.php` class altogether---class is sometimes seen as confusing for new users who think they are including the entire framework or don't understand the purpose of the class
- Only one class is required – `Zend/Loader.php` so the bootstrap will not grow.

### The disadvantages

- May break backwards compatibility for applications built before 0.9 (beta).
- If additional functions are needed, then need to require\_once additional classes.

----- Original Message -----

Subject: Re: [fw-general] Request for feedback: moving Zend.php to Zend/Zend.php  
Date: Mon, 26 Feb 2007 12:41:00 -0800  
From: Gavin Vess <gavin@zend.com>  
To: fw-general@lists.zend.com

I agree completely with Ralf, Simon, Matt R, and others regarding eliminating the current `Zend.php` class and splitting it up to achieve the flexibility and clarity the ZF has become famous for. In fact, I was pushing for something similar to this in an uncommitted version pre-0.7. Bill and I brainstormed about these kinds of changes before 0.7, but it was decided to delay consideration until a later date, in order to get 0.7 done sooner. I've always strongly disliked the old registry interface currently found in `Zend.php`. The creation of the newer `Zend_Registry` class was a move towards fixing the API without breaking BC. If we are going to clean things up in time for ZF 1.0, I certainly vote to do so before ZF 0.9.

Basically, the idea was to have an ultra-light core for "`Zend.php`" (really just a minimalistic framework loader / "bootstrap" - not an application bootstrap), which would then include optional and non-optional classes necessary for using the ZF. However, these "pieces" would be grouped under a subdirectory (e.g. "bootstrap") instead of mixed in with everything else at the top-level.

## 2. References

- [Zend Core include path utilities - Aleksey V. Zapparov](#) - Example of functionality that has utility, but should remain optional, yet easy to access for those needing it.

### 3. Component Requirements, Constraints, and Acceptance Criteria

#### @TODO

Most requirements take the form of "foo will do ...." or "foo will not support ...", although different words and sentence structure might be used. Adding functionality to your proposal is requirements creep (bad), unless listed below. Discuss major changes with your team first, and then open a "feature improvement" issue against this component.

### 4. Dependencies on Other Framework Components

- `Zend_Exception`

### 5. Theory of Operation

The component is instantiated with a mind-link that ...

### 6. Milestones / Tasks

Describe some intermediate state of this component in terms of design notes, additional material added to this page, and / code. Note any significant dependencies here, such as, "Milestone #3 can not be completed until feature Foo has been added to ZF component XYZ." Milestones will be required for acceptance of future proposals. They are not hard, and many times you will only need to think of the first three below.

- Milestone 1: [design notes will be published here](#)
- Milestone 2: Working prototype checked into the incubator supporting use cases #1, #2, ...
- Milestone 3: Working prototype checked into the incubator supporting use cases #3 and #4.
- Milestone 4: Unit tests exist, work, and are checked into SVN.
- Milestone 5: Initial documentation exists.

If a milestone is already done, begin the description with "[DONE]", like this:

- Milestone #: [DONE] Unit tests ...

### 7. Class Index

- `Zend_Magic_Exception`
- `Zend_Magic` (factory class)
- `Zend_Magic_MindProbe`
- `Zend_Magic_MindProbe_Intent`
- `Zend_Magic_Action`
- `Zend_Magic_CodeGen`

### 8. Use Cases

UC-01

```
require_once 'Zend/Registry.php';
$registry = Zend_Registry::getInstance();
$registry['key'] = 'value';
```

## 9. Class Skeletons

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