

Zend_Tool_Framework - Ralph Schindler

<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_Tool_Framework Component Proposal

Proposed Component Name	Zend_Tool_Framework
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend_Tool_Framework
Proposers	Ralph Schindler
Zend Liaison	TBD
Revision	1.0 - 2 June 2008: Initial Draft. (wiki revision: 8)

Table of Contents

1. Overview
 2. References
 3. Component Requirements, Constraints, and Acceptance Criteria
 4. Dependencies on Other Framework Components
 5. Theory of Operation
- Goals
- The subsystems that make up Zend_Tool_Framework
- Zend_Tool_Framework_Provider
 - Zend_Tool_Framework_Provider_Interface
 - Zend_Tool_Framework_Provider_Registry
 - Zend_Tool_Framework_Manifest
 - Zend_Tool_Framework_Loader
 - Zend_Tool_Framework_Loader_IncludePathLoader
 - Zend_Tool_Framework_Endpoint
 - Zend_Tool_Framework_Provider's built in providers
- Definitions and Terms
6. Milestones / Tasks
 7. Class Index
 8. Use Cases
 9. Class Skeletons

1. Overview

Zend_Tool_Framework is a component that is a generalize RPC-style framework for building a system that is capable of exposing, dispatching, and responding to requests that facilitate anything in the realm of "pragmatic tooling".

2. References

- [http://en.wikipedia.org/wiki/Scaffold_\(programming\)](http://en.wikipedia.org/wiki/Scaffold_(programming))
- <http://www.djangoproject.com/documentation/django-admin/>

3. Component Requirements, Constraints, and Acceptance Criteria

- This component **will** provide a Command Line Interface endpoint.
- This component **will** be extensible for additional endpoint interfaces: RPC, Code based, or perhaps other IDE's
- This component **will not** provide any project management utilities.
- This component **will not** provide any means of templating or code generation

4. Dependencies on Other Framework Components

- Zend_Console_Getopt
- PHP's Reflection
- Zend_Loader
- Zend_Version

5. Theory of Operation

Goals

Zend_Tool_Framework is a framework that provides:

- common interfaces, abstracts that allow will allow developers (through the most minimal of efforts) to create functionality and capabilities that are dispatchable by "tooling clients" (2).
- Implementation of "endpoints" (5) to be used to connect "tooling clients" to the Zend_Tool_Framework, the primary endpoint implementation being that of the "Command Line Interface Endpoint" (6)
- Implement interfaces that will identify classes as "Provider" or "Manifest" classes that can be utilized by the tooling system.
- Implement a standard set of "system/help providers" (8) that allow the system to report what the full capabilities of the system are as well as provide useful feedback. This might also be known as a "Help System".

In a more general sense, Zend_Tool_Framework might be useful in creating a system that can expose common functionalities such as the creation of files, the creation of project scaffolds, building of indexes, ... the list goes on. Central to all of the tasks of tooling is a resounding theme:

"Dont work harder, work smarter."

The subsystems that make up Zend_Tool_Framework

Zend_Tool_Framework_Provider

Zend_Tool_Framework_Provider_Interface

Zend_Tool_Framework_Provider represents the functional or "capability" aspect of the Zend_Tool_Framework framework. Fundamentally, Zend_Tool_Framework_Provider will provide the interfaces necessary to produce "providers", or bits of tooling functionality that can be called and used inside the Zend_Tool_Framework toolchain. The simplistic nature of implementing this "provider interface" allows the developer a "one-stop-shop" of adding functionality/capabilities to the Zend_Tool_Framework.

The provider interface is an empty interface and enforces no methods (this is the Marker Interface pattern):

```
<?php

interface Zend_Tool_Framework_Provider_Interface
{
}
```

For example, if a developer wants to add the capability of showing the version of a datafile that his 3rd party component is working from, there is only one class the developer would need to implement. Assuming the components is called `Some3rdParty_Component`, he would create a class/file named `Some3rdParty_Component_MyZfTool` (`MyZfTool.php`). This class would implement `Zend_Tool_Framework_Provider_Interface` and the body of this file would only have to look like the following:

```
// file name MyVersionTool.php
class Some3rdParty_Component_MyVersionTool implements Zend_Tool_Framework_Provider_Interface
{
    public function show()
    {
        $version = Some3rdParty_Component_Datafile::getVersion();
        echo 'Version number is: '. $version;
    }
}
```

Given that code above, and assuming the developer wishes to access this functionality through the Command Line Tooling Client, the call would look like this:

```
#zf show my-version
Version number is: 4.0
```

Zend_Tool_Framework_Provider_Registry

The primary objective of the Provider Registry is to take providers into the registry and process them into dispatchable methods. Provides will generally be introduced into the registry by means of a Loader (more on this concept in a later section of this proposal).

The provider registry's purpose is two fold. First, the registry will take complete provider marked objects (marked by `Zend_Tool_Framework_Provider_Interface`), then break them down into a signature object (`Zend_Tool_Framework_Provider_Signature`) that is capable of describing (in more finite terms) what actionable methods each provider is capable of dispatching.

Providers are similar in nature to action controllers in that each public method defined within a provider will become an actionable method. Generally, methods should be a verb. For example, if we had a "Version" provider. We might have a `show()` method. This allows us to issue the command "SHOW the VERSION".

Providers are capable of allowing the developer to organize a provider classes methods into groups by means of defining "Provider Specialties". To define specialties, all a provider must have is a protected/private method named `$_specialties`. The contents of this property should be an array of names of the specialties inside the provider. For example, lets say you only wanted the Version provider to return the major version number part of the version string. To accomplish this, we would create a protected `$_specialties = array('MajorPart');` property, and also create a method named `showMajorPart()`. This will then create an additional signature that can be dispatched by an endpoint. This dispatchable method is equivalent to the following command: "SHOW the MAJOR-PART of the VERSION".

Specialties are implemented so that providers do not run out of naming room inside their action/method namespace. This allows you to dispatch (with the same action), a number of different specialties of a provider.

Zend_Tool_Framework_Manifest

In short, the Manifest shall contain specific or arbitrary metadata that is useful to any provider or endpoint, as well as be responsible for loading any additional providers into the provider registry.

To introduce metadata into the registry, all one must do is implement the empty `Zend_Tool_Framework_Manifest_Interface`, and provide a `getMetadatas()` method which shall return an array of objects that implement `Zend_Tool_Framework_Manifest_Metadata`.

Metadata objects are loaded (by a loader defined below) into the Manifest Registry (Zend_Tool_Framework_Manifest_Registry). Manifests will be processed after all Providers have been found and loaded into the provider registry. This shall allow Manifests to create Metadata objects based on what is currently inside the Provider Registry.

Zend_Tool_Framework_Loader

The purpose of the Registry Loader is to find Providers and Manifest files that implement either Zend_Tool_Framework_Provider_Interface or Zend_Tool_Framework_Manifest_Interface. Once these files are found by a loader, providers are loaded into the Provider Registry and manifest metadata is loaded into the Manifest Registry.

Zend_Tool_Framework_Loader_IncludePathLoader

By default, the Tooling framework will use an include path based loader to find files that might include Providers or Manifest Metadata objects. Zend_Tool_Framework_Loader_IncludePathLoader, without any other options, will search for files inside the include path that end in Manifest.php, Tool.php or Provider.php. Once found, they will be tested (by the load() method of the Zend_Tool_Framework_Loader_Abstract) to determine if they implement any of the supported interfaces. If they do, an instance of the found class is instantiated, and it is appended to the proper registry.

Zend_Tool_Framework_Endpoint

Zend_Tool_Framework_Endpoint is the externally exposed interface that consumers are expected to interact with the Zend_Tool_Framework system through. The first endpoint executed shall be a command line interface. The Endpoint shall implement a static main() method such that will setup a default instance of the endpoint being used. The endpoint will be responsible for taking any environment or user supplied information, parsing it, and dispatching an actionable method on a provider. The output from the provider will then be provided to the endpoint implementation to send back to the user in a format that makes sense to the endpoint's implementation.

Zend_Tool_Framework_Provider's built in providers

These providers will be able to allow the system to reflect back to the user all of the capabilities and providers that are loaded within the tooling system.

- Version

```
<?php

require_once 'Zend/Version.php';

/**
 * Version Provider
 *
 */
class Zend_Tool_Framework_System_Provider_Version implements Zend_Tool_Framework_Provider_Interface
{

    const MODE_MAJOR = 'major';
    const MODE_MINOR = 'minor';
    const MODE_MINI = 'mini';

    protected $_specialties = array('MajorPart', 'MinorPart', 'MiniPart');

    /**
     * Show Action
     *
     * @param string $mode The mode switch can be one of: major, minor, or mini (default)
     * @param bool $nameIncluded
     */
    public function show($mode = self::MODE_MINI, $nameincluded = true)
    {
```

```

        $versionInfo = $this->_splitVersion();

        switch($mode) {
            case self::MODE_MINOR:
                unset($versionInfo['mini']);
                break;
            case self::MODE_MAJOR:
                unset($versionInfo['mini'], $versionInfo['minor']);
                break;
        }

        $output = implode('.', $versionInfo);

        if ($nameIncluded) {
            $output = 'Zend Framework Version: ' . $output;
        }

        echo $output;
    }

    public function displayAction()
    {
        $this->show();
    }

    public function showMajorPart($nameIncluded = true)
    {
        $versionNumbers = $this->_splitVersion();
        echo (($nameIncluded == true) ? 'ZF Major Version: ' : null) . $versionNumbers['major'];
    }

    public function displayMajorPart($nameIncluded = true)
    {
        $versionNumbers = $this->_splitVersion();
        echo (($nameIncluded == true) ? 'ZF Major Version: ' : null) . $versionNumbers['major'];
    }

    public function showMinorPart($nameIncluded = true)
    {
        $versionNumbers = $this->_splitVersion();
        echo (($nameIncluded == true) ? 'ZF Minor Version: ' : null) . $versionNumbers['minor'];
    }

    public function showMiniPart($nameIncluded = true)
    {
        $versionNumbers = $this->_splitVersion();
        echo (($nameIncluded == true) ? 'ZF Mini Version: ' : null) . $versionNumbers['mini'];
    }

    protected function _splitVersion()
    {
        list($major, $minor, $mini) = explode('.', Zend_Version::VERSION);
        return array('major' => $major, 'minor' => $minor, 'mini' => $mini);
    }

```

```
}  
}
```

Definitions and Terms

- (1) Zend_Tool_Framework - The framework which exposes tooling capabilities.
- (2) Tooling Client - A developer tool that connects to and consumes "Zend_Tool_Framework".
- (3) Command Line Tool / Command Line Interface / zf.php - The "tooling client" for the command line.
- (4) Zend Studio - An IDE based "tooling client", connects via the RPC "Zend_Tool_Framework_Endpoint"
- (5) Endpoint - The subsystem of "Zend Tool" that exposes an interface such that "tooling clients" can connect, query and execute commands.
- (6) Cli Endpoint - An endpoint implementation responsible for exposing "Zend Tool RPC" via the "command line interface".
- (7) Rpci Endpoint - An endpoint implementation responsible for exposing "Zend Tool RPC" via the "remote procedure call interface".
- (8) Provider - A subsystem and a collection of built-in functionality that the "Zend Tool" system exports.
- (9) ZFProject Provider - A set of providers specifically for creating and maintaining Zend Framework based projects.
- (10) Manifest - A subsystem for defining, organizing, and disseminating "provider" requirement data.

6. Milestones / Tasks

- 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_Tool_Framework_Endpoint
Zend_Tool_Framework_Endpoint_Abstract
Zend_Tool_Framework_Endpoint_Dispatcher
Zend_Tool_Framework_Endpoint_Exception
Zend_Tool_Framework_Endpoint_Request
Zend_Tool_Framework_Endpoint_Response
Zend_Tool_Framework_Loader
Zend_Tool_Framework_Loader_Abstract
Zend_Tool_Framework_Loader_Chain
Zend_Tool_Framework_Loader_IncludePathLoader
Zend_Tool_Framework_Loader_IncludePathLoader_RecursiveFilterIterator
Zend_Tool_Framework_Loader_IncludePathLoader

Zend_Tool_Framework_Manifest
Zend_Tool_Framework_Manifest_ActionMetadata
Zend_Tool_Framework_Manifest_Interface
Zend_Tool_Framework_Manifest_Metadata
Zend_Tool_Framework_Manifest_ProviderMetadata
Zend_Tool_Framework_Manifest_Registry
Zend_Tool_Framework_Provider
Zend_Tool_Framework_Provider_Action
Zend_Tool_Framework_Provider_Interface
Zend_Tool_Framework_Provider_Registry
Zend_Tool_Framework_Provider_Signature
Zend_Tool_Framework_System
Zend_Tool_Framework_System_Action
Zend_Tool_Framework_System_Action_Create
Zend_Tool_Framework_System_Action_Delete
Zend_Tool_Framework_System_Manifest
Zend_Tool_Framework_System_Provider
Zend_Tool_Framework_System_Provider_Providers
Zend_Tool_Framework_System_ProviderVersion

8. Use Cases

UC-01

Example of extending the tooling systems capabilities

```
<?php

// name will be taken as Phpinfo
class My_PhpinfoTool implements Zend_Tool_Framework_Provider_Interface
{

    // action is "show"
    public function showAction()
    {
        phpinfo();
    }

}

// this class is now exposed to the tooling system b/c it name matchs "**Tool" and it implements the
Zend_Tool_Framework_Provider_Interface interface
```

9. Class Skeletons

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