Zend_Console - Wil Sinclair

<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_Console Component Proposal

Proposed Component Name	Zend_Console
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend_Console
Proposers	Wil Sinclair
Revision	1.0 - 20 December 2007: Proposal created. (wiki revision: 23)

Table of Contents

1. Overview

Zend_Console provides classes to implement a command-line interface (CLI) and a system executable that loads the appropriate classes and executes the user's command. Any component may use Zend_Console interfaces to create custom commands. The classes in Zend_Console surface Zend_Build actions and resources (see the Zend_Build proposal) for configuration and execution on the command line. This component also introduces the concept of 'console context' which allows options to apply to specific parts of the command depending on where they occur in the command string. In addition Zend_Console will provide a consistent help system that utilizes the documentation in action and resource manifest files. This component will also facilitate scripting by supporting a fully non-interactive mode and returning relevant error codes. Optionally, the component will support multiple levels of verbosity and an interactive mode to collect options from the user in a question-answer loop. This CLI will be the primary interface to the Zend_Build component for the forseeable future.

2. References

- Ruby on Rails
- Rails Command Line Reference
- Symfony Generators

3. Component Requirements, Constraints, and Acceptance Criteria

- This component **must** provide a system executable for performing actions on resources in a project or the project itself in a command line shell (as in 'zf create project' executed at the Windows command prompt).
- This component **must** load action and resource classes from any component in the application's library to facilitate adding custom actions and resources to the CLI system without extra configuration. The system **must** locate manifest files located within the components package structure to identify such classes.
- This component **must** read options that are global-, action-, or resource-scoped in Gnu getopt or Zend option syntax.
- This component **must** support one single-word name (underscores allowed) and one short (one- or two-letter) alias for all CLI-enabled actions and resources.
- This component **must** support simple execution from scripts with a fully non-interactive mode.
- This component **must** return a non-zero error code to the shell if an error has occured.
- This component **must** return an error code or zero to the shell on successful completion of the command.
- This component **should** return a well-defined error code to the shell that indicates the error type.
- This component **should** support optional execution in interactive mode to prompt for options.
- All CLI commands **should** be atomic where possible.
- Pluralization of resource types on the command line would be **nice to have** to improve readability of commands for English speakers, but singular syntax **must** be supported.
- Developers **must** be able to specify the project profile to use in the CLI (see Zend_Build).

4. Dependencies on Other Framework Components

- Zend_Exception
- Zend_Console_Getopt
- Zend_Build
- Zend_Config
- Zend_Loader
- Zend_Log

5. Theory of Operation

The basic format of the CLI is:

zf <global options> <action> <action options> <resource> <resource options>

'zf' is the system executable. 'zf options' are the options that apply globally. 'action' is the action that this command must perform. 'action options' are options that apply only to the 'action' specified. 'resource' is the resource that the action is performed upon. 'resource options' are options that apply only to the 'resource' specified. All options can be written in GNU getopt or Zend option format. Refer the to the Zend_Console_GetOpt component for full option syntax. A 'command' represents the entire command string and can optionally have an action and a resource. The command must have an action if it has a resource, although it can have an action and no resource. The 'zf' command must support verbosity and help options. If the help option is passed, the 'zf' command will provide a usage statement that reflects the usage rules above. All actions and resources must also accept the help option.
Actions and resources can be added to the library by first implementing Zend_Build_Action_Interface and Zend_Build_Resource_Interface, respectively. Next, the developer must add one manifest file per action or resource to his/her component with the name '<classname>-ZFManifest.*'. These manifest files can be any file type supported by Zend_Config, but must contain all information about the resource for console usage, including all options and help information. This configuration file may also contain information required by Zend_Build. The following is an example of such a file in the xml format:

```
<?xml version="1.0"?>
<configdata>
    <command_context type = 'resource' name = 'library_dir' alias = 'ld'
class='Zend_Build_Resource_LibraryDirectory'>
        <attribute getopt='apple|a'>
            <usage>
                This is how you use apple with library_dir.
            </usage>
        </attribute>
        <attribute getopt='banana|b=i'>
            <usage>
                This is how you use banana with library_dir: banana must have an
integer param.
            </usage>
        </attribute>
        <attribute getopt='pear|p-s'>
            <usage>
                This is how you use pear with library_dir: pear can have a string
param.
            </usage>
        </attribute>
     </command_context>
</configdata>
```

At command execution these files will be detected in all components on the include_path and loaded as necessary.
Each command executes within the context of a project. The metadata required by the command to perform operations on this project are stored in a 'project profile' (see the Zend_Build proposal). A command can read, add, delete, or update data in this profile. If a file is found at $PROJECT_ROOT/profile.xml, the command will use this profile; otherwise, the default profile will be found in $PROJECT_ROOT/Zend/Build/profiles/simple-profile.xml, which represents a basic Zend Framework project with MVC architecture and no modules. Additional profiles will ship with the release of this component, but the full list of profiles is TBD. The following is an example of such a profile for a simple Zend Framework project:

```
<?xml version="1.0"?>
<configdata>
<project name="basic_mvc_project">
  <application_dir name="application">
    <controller_dir name="controllers"/>
    <model_dir name="models"/>
    <view_dir name="views"/>
  </application_dir>
  <data_dir name="data"/>
  <public_dir name="htdocs"/>
  <library_dir name="lib"/>
  <trash_dir maxSize="100" name="trash"/>
</project>
</configdata>
```

Zend_Config will be used to serialize and deserialize this project profile. As an example, if the developer were to run a command 'zf create controller Blog', the component would first look up the project profile (in this case using the default at $PROJECT_ROOT/profiles/simple-profile.xml), locate the Zend_Build_Resource_ControllerDirectory, add the controller template with the correct class and file name to this directory, and finally add an element of type Zend_Build_Resource_Controller with the name 'Blog' as a child of the Zend_Build_Resource_Directory. Likewise a command to delete the controller will locate the Zend_Build_Resource_ControllerDirectory, delete the controller class file, and remove the child element corresponding to this controller from the Zend_Build_Resource_ControllerDirectory element. This project profile may also be accessed by other tools, such as Zend Studio, to retrieve metadata about the project.
This component will add a directory named 'profiles' to the top-level directory of the distribution. This directory will contain all project profiles and file template necessary to generate a small number of common project structures.

This component will also add a directory named 'bin' to the root directory of the Zend Framework distribution. This directory will contain a PHP-CLI script named 'zf.php', a batch file named 'zf.bat', and a sh shell script named 'zf.sh'. The developer will only be required to add this directory to their system executable path to access all of the functionality described in this document.

6. Milestones / Tasks

- Milestone 1: [DONE] Draft proposal submitted for community review
- Milestone 2: Working prototype checked in to incubator for community evaluation
- Milestone 3: Proposal finalized
- Milestone 4: All unit test passing with coverage at 90% or higher for all code
- Milestone 5: Documentation finished in English

7. Class Index

- Zend_Console_Context_Interface
- Zend_Console_Context_Action
- Zend_Console_Context_Resource
- Zend_Console
- Zend_Console_ErrorCodes
- Zend_Console_Exception
- Zend_Console_Factory

8. Use Cases

UC-01

User creates a project

The user runs the following command:

```
zf create project my-project
```

The system looks for a profile in the current directory. None is found. The system searches for the 'profiles' directory in the Zend Build package (if more than one instance of the Zend Framework library is on the classpath, it searches them in order. The system creates a project from the simple-profile.xml project profile and adds a symlink to the Zend Framework library where the 'profiles' folder is located in the location specified by the Zend_Resource_LibraryDirectory resource.

UC-02

User creates a new action 'touch'

User creates class Zend_MyComponent_Touch implementing Zend_Build_Action_Interface as follows:

```
class Zend_MyComponent_Touch extends Zend_Build_Action_Abstract implements
{
    // Insert code here
}
```

User then creates a file called zf-manifest.ini in the directory Zend/MyComponent:

```
<?xml version="1.0"?>
<configdata>
    <command_context type = 'action' name = 'touchlibrary_dir' alias = 't'
class='Zend_MyComponent_Touch'>
        <attribute getopt='apple|a'>
            <usage>
                This is how you use apple with library_dir.
            </usage>
        </attribute>
        <attribute getopt='banana|b=i'>
            <usage>
                This is how you use banana with library_dir: banana must have an
integer param.
            </usage>
        </attribute>
        <attribute getopt='pear|p-s'>
            <usage>
                This is how you use pear with library_dir: pear can have a string
param.
            </usage>
        </attribute>
     </command_context>
</configdata>
```

System will then automatically load touch as needed and run the command on the specified resource.

9. Class Skeletons

```
h3. Zend_Console_Context_Interface
interface Zend_Console_Context_Interface
{
    public function __construct(array $argv = array());
    public function getUsage();
    public function getOptions();
}

h3. Zend_Console_Context_Action
class Zend_Console_Context_Action
{
    public function __construct(array $argv = array());
    public function getUsage();
    public function getOptions();
```

```
}


h3. Zend_Console_Context_Resource
class Zend_Console_Context_Resource
{
    public function __construct(array $argv = array());
    public function getUsage();
    public function getOptions();
}

h3. Zend_Console_Command
class Zend_Console_Command implements Zend_Console_Context_Interface
{
 public function init(array $argv = array(), $verbosity = 0);
 public function getUsage();
 public function getOptions()

        public function getAction();
        public function getResource();
}

h3. Zend_Console_ErrorCodes
interface Zend_Console_ErrorCodes
{
    const CATCH_ALL                     = 1;
    const ACTION_NOT_FOUND              = 2;
    const RESOURCE_NOT_FOUND            = 3;
    const INVALID_ACTION_CLASS          = 4;
    const INVALID_RESOURCE_CLASS        = 5;
}

h3. Zend_Console_Exception
class Zend_Console_Exception extends Zend_Exception
{
    public function __construct($consoleMessage = null, $consoleUsage = null,
$consoleCode =1);
    public function getConsoleMessage();
    public function getConsoleUsage();
    public function prependUsage($str);
    public function getConsoleCode();
}

h3. Zend_Console_Factory
class Zend_Console_Factory
{
    public static function makeConsoleCommand($argv);
    public static function makeConsoleAction($argv);
    public static function makeConsoleResource($argv);
}

h3. zf.php (PHP CLI script called from the shell)
#!/usr/bin/php
<?php
/**
 * Zend Framework
 *
 * LICENSE
 *
```

```
 * This source file is subject to the new BSD license that is bundled
 * with this package in the file LICENSE.txt.
 * It is also available through the world-wide-web at this URL:
 * http://framework.zend.com/license/new-bsd
 * If you did not receive a copy of the license and are unable to
 * obtain it through the world-wide-web, please send an email
 * to license@zend.com so we can send you a copy immediately.
 *
 * @category   Zend
 * @copyright  Copyright (c) 2005-2007 Zend Technologies USA Inc.
(http://www.zend.com)
 * @license    http://framework.zend.com/license/new-bsd     New BSD License
 */

require_once('Zend/Console/Factory.php');

function println($str = '')
{
    print("$str\n");
}

try {
    // Initiate all arguments and validate them
    $command = Zend_Console_Factory::makeConsoleCommand($_SERVER['argv']);

    if($command) {
        // Looks like everything is in order, execute the command
        $command->execute();
    }
}  catch (Zend_Console_Exception $e) {
    println($e->getConsoleMessage());
    println();
    println('Usage:');
    println($e->getConsoleUsage());
    exit($e->getConsoleCode());
}

// As far as we know, everything came off just fine
```

```
exit(0);
?>
```

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