Zend_Application Module Configuration - Pádraic Brady, Rob Zienert

<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_Application Module Configurators Component Proposal

Proposed Component Name	Zend_Application Module Configurators
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend_Application Module Configurators
Proposers	Pádraic Brady Rob Zienert
Revision	1.0 - 14 September 2009 (wiki revision: 6)

Table of Contents

1. Overview
The Problem Module Configurators Help Solve
What is a Configurator?
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
4. Dependencies on Other Framework Components
5. Theory of Operation
6. Milestones / Tasks
7. Class Index
8. Use Cases
9. Class Skeletons

1. Overview

The Problem Module Configurators Help Solve

In Zend_Application, one can define a "Resource" as a package of instructions designed to initialise and configure an application object whether it be Zend_Layout, Zend_Db or some other class. These Resources are all executed prior to the routing of requests. Since no routing has occurred, Zend_Application based bootstrap classes initialise all Resources before any identification of the current controller, action and module can be completed. As a result, these initial Resources are non-specific and by default apply to the entire application. This extends to the methods of configuration used. The typical application.ini contains settings which apply to these application wide Resources. One can add module prefixed configuration options, but these are applied to the existing Resources in a confusing way. A module prefixed resource setting will overwrite any preceeding non-prefixed resource setting.

The result is that module-specific configuration, i.e. configuration which is applied only when a module is accessed, is not currently possible without introducing a custom set of front controller plugins whose preDispatch() or routeShutdown() methods are used to detect the current module, and apply such configuration. This has led to any number of variants on this strategy, the most common example being switching layouts depending on what module is being accessed. What they all have in common, however, is their lack of a comprehensive documented approach, making reusable modules capable of being distributed outside their host environment difficult to achieve.

This proposal addresses one particular facet of the reusability problem - module specific on-access configuration. It does relate to merging application-wide module configurations though this may be appended to the proposal or added to a separate proposal in the future.

What is a Configurator?

A Configurator is a class that exists parallel to an existing Resource class. For example, Zend_Application_Resource_Layout would be complemented by Zend_Application_Module_Configurator_Layout. While a Resource class is concerned with instantiating new objects and imposing an initial configuration for the application as a whole, a Configurator is concerned with two additional tasks:

1. To reconfigure existing Resources with a module specific configuration; or
2. To bootstrap Resources needed by a specific module which were not bootstrapped beforehand, i.e. lazy loading.

The Configurator system operates by running after routing has taken place but before the request has been dispatched. Using a front controller plugin, it detects the current module and searches for a module-specific configuration file. Using this new configuration, it utilises concrete Configurator classes to modify or initialise new Resources as needed. The system is kept minimal by delegating work as needed to the bootstrap class registered to the front controller.

As a convention, it is assumed (for now) that on-access configuration files will be stored by modules in the relative path /configs/module.ini.

2. References

Proof Of Concept Implementation

3. Component Requirements, Constraints, and Acceptance Criteria

- MUST allow for on-access configuration and initialisation of Module Resources
- MUST impose a standard and strict convention for configuration loading
- MUST implement Configurators for any Resource capable of being reconfigured after routing
- MUST allow the setting of custom Configurator classes
- MAY allow for Configurators to be defined as methods on a class
- MAY allow for application level merging of module configurations used in initial bootstrapping

4. Dependencies on Other Framework Components

Zend_Application
Zend_Application_Bootstrap_Bootstrapper
Zend_Application_Resource_ResourceAbstract
Zend_Config
Zend_Controller_Plugin_Abstract

5. Theory of Operation

h5 Component Classes and Workflow

There are three parts to the Configurator solution as currently designed:

1. A front controller plugin, Zend_Controller_Plugin_ModuleConfigurator, implements a routeShutdown() method which is called when routing has completed. The plugin uses the discovered module name to check whether or not a module-specific configuration file exists. If it does, the configuration is loaded up and passed to an instance of Zend_Application_Module_Configurator. This Configurator class is then executed by calling its run() method.

2. Zend_Application_Module_Configurator accepts a module specific configuration and an instance of the bootstrap class registered to the front controller. It assesses the configuration to compile a list of named Resources that the current module needs to either modify or initialise. This list is then traversed. For each Resource noted, a matching Configurator object is instantiated using the registered plugin loader. The Resource configuration and bootstrap instance are passed to this Configurator object, and the object's init() method is called. An exception is thrown if a matching Configurator cannot be located. The class uses a plugin loader to allow the setting of custom Configurators.

3. Each Configurator class will attempt one of two options. If the bootstrap object contains a pre-initialised Resource, then the Configurator will attempt to reconfigure the object it represents using any available setters. Where possible, the Resource object's setOption() method will be used if it exists (this merely minimises the code needed for such a simple task). If the bootstrap object does not have the named resource, the Configurator will pass the module-specific configuration into the bootstrap object and request that it bootstrap the Resource needed. Both options should require a minimum of code since the bootstrap does most of the work itself.

4. [Suggested] Similar to Resources, Configurators could also be added to a class as discrete methods. For example, a Layout Configurator could be defined as the method _initLayout() in a class stored in the Module's root directory as Configurator.php. This is an optional feature to be discussed, and will not be required as Module Bootstraps are.

6. Milestones / Tasks

- Milestone 1: Publish and have proposal approved with Zend comments to resolve
- Milestone 2: Complete initial component with unit tests
- Milestone 3: Complete testing against a selection of web-based/desktop feed readers
- Milestone 4: Verify that code operates under PHP 5.3/6.0-dev
- Milestone 5: Complete documentation

7. Class Index

- Zend_Controller_Plugin_ModuleConfigurator
- Zend_Application_Module_Configurator
- Zend_Application_Module_Configurator_ConfiguratorAbstract
- Zend_Application_Module_Configurator_Layout
- Zend_Application_Module_Configurator_Db
- Zend_Application_Module_Configurator_Locale
- Zend_Application_Module_Configurator_Translate
- Zend_Application_Module_Configurator_View

Note: Resources like FrontController and Router are omitted since they probably should not be altered after routing has occurred. Modifying these on a per-module basis must be done before Configurators kick in so it is handled by the application configuration file, or a system for merging module configuration files from another location than the convention used herein. This will be addressed in another proposal, or appended here as a separate proposed solution.

8. Use Cases

9. Class Skeletons

Please see proof of concept repository for all classes to date. As a test-driven design approach is utilised a complete set of class skeletons does not yet exist.

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