

MVC Reorganization Proposal - Matthew Weier O'Phinney

<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 MVC Components Component Proposal

Proposed Component Name	Zend MVC Components
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend MVC Components
Proposers	Matthew Weier O'Phinney
Revision	1.1 - 22 September 2006: Created (wiki revision: 12)

Table of Contents

1. Overview
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
4. Dependencies on Other Framework Components
5. Theory of Operation
 - Request Objects
 - Zend_Request_Interface
 - Zend_Http_Request
 - Zend_Controller_Request_Abstract
 - Zend_Controller_Request_Http
 - Router changes
 - Dispatcher changes
 - Zend_Controller_Dispatcher_Interface
 - Zend_Controller_Dispatcher
 - Zend_Controller_Front
 - Changes necessary
 - Zend_Controller_Action
6. Milestones / Tasks
7. Class Index
8. Use Cases
9. Class Skeletons

1. Overview

There are many proposed changes. The most important and far-reaching is the creation of a variety of Request classes and interfaces that will be used to inform the entire controller workflow of the request environment. This change will allow better unit testing of components (as the controllers are no longer dependent on the web environment in which they reside); the ability for the various objects in the controller chain to pass information to one another; and the ability to use the controller chain within non-web environments, such as the CLI or a GUI.

Some discussion centered around removing the need for the Router and/or Dispatcher. We have decided to keep these, but have them receive and return Request objects as the common currency instead of the Dispatcher Token. By doing so, we have a common environment that all objects share during execution.

Another discussion regarded the optional passing of a Zend_Registry object through the controller chain. However, nobody stepped forth to show (1) a solid use case where this would be necessary, and (2) a reason this couldn't be done via other means. One method, proposed by Michael Sheakoski and revised by Simon Mundy, would add a method to the Front Controller to allow passing optional invocation parameters to other objects and method calls in the controller dispatch chain:

```
$controller->addParam($registry); // single parameter
$controller->addParam(array($registry, $acl)); // multiple parameters
```

At this stage, any parameter added in this way will be passed to the constructors for the Router, Dispatcher, and Action Controller objects; additionally, Zend_View_Helper will have a similar method for passing parameters to helper scripts. This methodology will help the API remain simple, while adding the flexibility and power desired for more advanced use cases.

2. References

- [Changes to Zend MVC](#)
- [Zend Controller Front in a Subdirectory](#)
- [Zend_Http_Request Proposal \(Sheakoski\)](#)
- [Zend_Http_Request Proposal \(Philip\)](#)

3. Component Requirements, Constraints, and Acceptance Criteria

- Allow Front Controller to work from a subdirectory easily
- Allow for developer-defined routing easily
- Provide tools and infrastructure for creating custom routing and dispatching
- Provide methods for pushing data from the Front Controller through the controller chain
- Decouple the MVC components from a web-only paradigm by allowing requests to come from any environment so long as they provide the controller and action
- Allow basic usage via a one-liner, as well as more advanced usage

4. Dependencies on Other Framework Components

- [Zend_Request_Interface](#)
- [Zend_Http_Request](#) (optional)

5. Theory of Operation

Operation should allow for a simple static method that will create a default request object to push down through the controller chain, as well as advanced usage that accommodates custom routers, dispatchers, and request objects.

The following changes will need to be made.

Request Objects

The various members of the MVC working group identified the need for a Request object early. The rationale for it is to provide encapsulation of common functionality that would be needed for routing and dispatching, and to provide a replaceable object that can be used during testing on the CLI (allowing for flexible unit testing approaches). Additionally, by having generic request inheritance, MVC components could be used for non-web environments such as the CLI or GUI.

Request objects are primarily data containers, and typically shouldn't manipulate the data (other than setting based on input).

Zend_Request_Interface

Generic request interface. All request classes would implement it – Zend_Http_Request, Zend_Controller_Request, Zend_Cli_Request, etc., are all candidates.

Basic functionality is simply as an OOP accessor.

Zend_Http_Request

Proxies the HTTP request environment, providing accessors for \$_GET, \$_POST, \$_COOKIE, \$_SERVER, \$_ENV, and PATH_INFO. Additionally, it contains methods accessors for the base URL and path.

Functionality will follow that of Mike Sheakoski's proposal. Additionally, it should provide functionality to get items out of the PATH_INFO by index and key (value would be from position(key) + 1)

Zend_Controller_Request_Abstract

Provides an interface of what needs to be provided to the controllers. Request objects used with the controller classes would need to implement this interface.

Defining such an interface allows developers to decouple their applications from the web, and could allow for use of the MVC components in CLI and GUI (e.g., PHP-GTK) environments.

Zend_Controller_Request_Http

Default Request object used by controller classes, this is an HTTP request. It extends the functionality of Zend_Http_Request by adding Controller specific methods (as defined in the Zend_Controller_Request_Abstract).

Router changes

Much of the functionality of the Router is now performed by the request. However, routing is beyond the scope of a Request object (which is a data container), and should have dedicated functionality. Such functionality may include regular expression matching of request parameters in order to determine the current route, using PATH_INFO indices to determine controller and action, etc. It would set the controller and action in the request and return the request object.

Dispatcher changes

Part of the discussion has centered around the idea of eliminating the Router and Dispatcher. As noted in the discussion of the Router and Request objects, above, each has a specific domain in which they work. In the case of the Dispatcher, keeping it in a separate class and following an interface allows some flexibility for advanced developers and/or special scenarios. One such scenario might be integrating legacy applications: a custom dispatcher could dispatch to legacy, procedural scripts instead of Action Controllers.

One common theme, however, is that the dispatcher would receive the Request object as its token instead of a Dispatcher Token. This allows the Dispatcher to have access to the entire request environment shared by all segments of the controller workflow. Zend_Controller_Dispatcher-Token and Zend_Controller_Dispatcher-Token-Interface will be removed entirely.

Zend_Controller_Dispatcher_Interface

Instead of receiving or returning dispatch tokens, Request objects will be used for the same purpose and more.

Zend_Controller_Dispatcher

The Dispatcher needs to be re-tooled to use the Request object. Additionally, instead of calling the Action Controller's run() method, it will perform

actions similar to below:

```
$params          = $this->getParams();
array_unshift($params, $request);

$curController = $this->formatControllerName($request->getControllerName());
$curAction     = $this->formatActionName($request->getActionName());
$reflection    = new ReflectionClass();
$controller    = $reflection->newInstanceArgs($params);

if ($controller->preDispatch($request)) {
    // Return true to indicate another action is being called and the current
    // action skipped
    return true;
}

// Return true to indicate another action is being called; false to end
// dispatching
$action = $reflection->getMethod($curAction);
return $action->invokeArgs($controller, $params) ||
    $controller->postDispatch($request);
```

Zend_Controller_Front

Current issues with the Front Controller include:

- Action controllers have no knowledge of the request environment
- Plugins, dispatchers, and routers do not share the same request information and thus are decoupled to the point of being crippled
- No ability to pass extra custom parameters to the action controllers, dispatcher, or router

Changes necessary

- Add methods
 - addParam()
- Modify dispatch() method
 - Allow passing optional Zend_Controller_Request_Abstract object
 - Pass optional arguments to router, dispatcher
 - If optional arguments set via passToAction(), pass to dispatcher::passToAction()
 - Rework to utilize Zend_Controller_Request_Abstract object throughout dispatch process

Zend_Controller_Action

Current issues include:

- No ability to do pre/post action items using the same controller instance
- No knowledge of the request environment
- No ability for a pre/post action to define the next action to call.

The proposed changes would remove the run() method from the dispatch loop to instead call the action directly. However, it would bookend this call with calls to preDispatch() and postDispatch() on the action controller, if defined. The postDispatch() method may perform extra tasks prior to passing on handling to the dispatch loop; by modifying the Request object and setting its dispatched flag to false, it may also override what additional actions may be called in the dispatch loop. The preDispatch() method, by modifying the Request object and resetting its dispatched flag, can cause the dispatch loop to skip the current action and move on to the one now specified in the Request object.

An action may specify an additional action to perform by modifying the Request object and resetting its dispatched flag.

The constructor would receive the Request object at instantiation. Finally, the actions may receive extra parameters if the front controller and/or dispatcher has been passed information via addParam(). These are stored in the \$_invokeArgs property, to which accessors will be provided.

In order to enforce the requirement that the constructor receive the Request object, the constructor will be made final, and will set the protected \$_request property. A new init() method will be called as the final action during object instantiation and may be overridden by developers to provide custom constructor functionality.

run() would be retained for backwards compatibility and to allow usage of the Action Controller as a Page Controller. It will be made non-final, and will do the following:

```
public function run(Zend_Controller_Request_Abstract $request)
{
    $this->preDispatch($request);
    $action = $request->getActionName() . 'Action';
    $this->{$action}($request);
    $this->postDispatch($request);
}
```

6. Milestones / Tasks

1. Creation of Request interfaces and objects
2. Retool Controllers (Front, Action), Dispatchers, and Routers to use Request objects; rework logic to accommodate.
3. Testing; full unit test coverage
4. Sync documentation with source changes

7. Class Index

- Zend_Request_Interface
- Zend_Http_Request implements Zend_Request_Interface
- Zend_Controller_Request_Abstract
- Zend_Controller_Request_Http extends Zend_Http_Request implements Zend_Controller_Request_Abstract
- Zend_Controller_Router_Interface
- Zend_Controller_Router
- Zend_Controller_RewriteRouter
- Zend_Controller_Dispatcher_Interface
- Zend_Controller_Dispatcher
- Zend_Controller_Action

8. Use Cases

UC-01

```
// Basic usage
require_once 'Zend/Controller/Front.php';
Zend_Controller_Front::run('/path/to/controllers');
```

UC-02

```

// Advanced usage
require_once 'Zend/Controller/Front.php';
require_once 'Zend/Controller/Request.php';
require_once 'My/Router.php';
require_once 'My/Dispatcher.php';

$controller = new Zend_Controller_Front();
$request     = new My_Router();
$router      = new My_Router();
$dispatcher  = new My_Dispatcher();

$dispatcher->setControllerPath('/my/custom/controllers');
$request->setBaseUrl('/some/subpath');
$request->setBasePath('/my/html');

$controller->setDispatcher($dispatcher);
$controller->setRouter($router);
$controller->dispatch($request);

```

9. Class Skeletons

```

interface Zend_Request_Interface
{
    public function __get($key);
    public function __set($key, $value);
    public function __isset($key);
    public function get();
    public function set();
    public function has();
    public function getParam($key);
    public function setParam($key, $value);
    public function getParams();
    public function setParams($params);
}

class Zend_Http_Request implements Zend_Request_Interface
{
    public function isPost();
    public function getMethod();
    public function getQuery();
    public function setQuery();
    public function getPost();
    public function setPost();
    public function getPathInfo();
    public function setPathInfo();
    public function getAlias();
    public function setAlias();
    public function getAliases();
    public function getRequestUri();
    public function setRequestUri();
    public function getBaseUrl();
    public function setBaseUrl();
}

```

```

    public function getBasePath();
    public function setBasePath();
    public function getCookie();
    public function getServer();
    public function getEnv();
}

abstract class Zend_Controller_Request_Abstract
{
    public function getControllerName();
    public function setControllerName($value);
    public function getActionName();
    public function setActionName($value);
    public function getParam($key);
    public function setParam($key, $value);
    public function getParams();
    public function setParams($array);
    public function setDispatched($flag = true);
    public function isDispatched();
}

class Zend_Controller_Request_Http extends Zend_Controller_Request_Abstract
{
    /**
     * Zend_Http_Request object
     * protected $_request;

    /**
     * Proxy to Zend_Http_Request
     * public function __call($method, $args);
     * public function __get($key);
     * public function __set($key, $value);

    public function getControllerKey();
    public function setControllerKey();
    public function getActionKey();
    public function setActionKey();
}

abstract class Zend_Controller_Response_Abstract
{
    public function setHeader($key, $value, $append = true);
    public function getHeaders();
    public function clearHeaders();
    public function setBody($content);
    public function appendBody($content);
    public function __toString();
}

interface Zend_Controller_Router_Interface
{
    /**
     * Instead of receiving the Dispatcher as a parameter, a Request
     * object would be passed instead.
     */
    public function route(Zend_Controller_Request_Abstract $request);
}

interface Zend_Controller_Dispatcher_Interface

```

```

{
    public function formatControllerName($unformatted);
    public function formatActionName($unformatted);
    public function isDispatchable(Zend_Controller_Request_Abstract $request);
    public function dispatch(Zend_Controller_Request_Abstract $request);
    public function passToAction($controller, $action, $array);
}

abstract class Zend_Controller_Action
{
    /**
     * Extra parameters passed to the instantiator
     * @var array
     */
    protected $_invokeArgs = array();

    /**
     * @var Zend_Controller_Request_Abstract
     */
    protected $_request;

    /**
     * Constructor
     *
     * Receives request and response objects as sole required parameters, and
     * sets them in the object. Any additional arguments are stored in
     * $_invokeArgs and passed on as parameters to init().
     */
    final public function __construct(Zend_Controller_Request_Abstract $request,
        Zend_Controller_Response_Abstract $response)
    {
        $this->_request = $request;
        $this->_response = $response;

        if (1 < func_num_args()) {
            $argv = func_get_args();
            array_shift($argv); // remove request argument
            array_shift($argv); // remove response argument
            $this->_invokeArgs = $argv;
        }

        $reflection = new ReflectionObject($this);
        $init = $reflection->getMethod('init');
        $init->invokeArgs($this, $this->_invokeArgs);
    }

    public function getInvokeArgs()
    {
        return $this->_invokeArgs;
    }

    /**
     * Receives all arguments passed to constructor except request
     */
    public function init() {}

    public function preDispatch() {}

    public function postDispatch() {}
}

```

```

/**
 * _forward to new action
 *
 * Sets new controller name, action, and params, and resets request
 * dispatched flag to false, indicating new actions need to be processed.
 */
protected function _forward($controller, $action, $params = null())
{
    $this->_request->setControllerName($controller);
    $this->_request->setActionName($action);
    $this->_request->setParamsName($params);
    $this->_request->setDispatched(false);
}
}

class Zend_Controller_Front
{
    public static function run($path);

    public function dispatch(Zend_Controller_Request_Abstract $request = null,
Zend_Controller_Response_Abstract $response = null);

    // Add parameter(s) to pass to router, dispatcher, and action
    // constructor/actions
    public function addParam($scalar | $array);
    public function getParams();

    public function setRequest();
    public function getRequest();
    public function setResponse();
    public function getResponse();
}

// Sample idea for dispatch loop in Zend_Controller_Front
do {
    $request->setDispatched(true);

    // notify plugins that a dispatch is about to occur
    $this->_plugins->preDispatch($request);

    // skip requested action if preDispatch() has reset it
    if (!$request->isDispatched()) {
        continue;
    }

    $response = $this->getDispatcher()->dispatch($request, $response);

    // notify plugins that the dispatch has finish
    $this->_plugins->postDispatch($request);
} while (!$request->isDispatched());

```



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