Table Of Contents

Previous topic

Action Controllers

Next topic

The Response Object

Action Helpers

Introduction

Action Helpers allow developers to inject runtime and/or on-demand functionality into any Action Controllers that extend Zend_Controller_Action. Action Helpers aim to minimize the necessity to extend the abstract Action Controller in order to inject common Action Controller functionality.

There are a number of ways to use Action Helpers. Action Helpers employ the use of a brokerage system, similar to the types of brokerage you see in Zend_View_Helper, and that of Zend_Controller_Plugin. Action Helpers (like Zend_View_Helper) may be loaded and called on demand, or they may be instantiated at request time (bootstrap) or action controller creation time (init()). To understand this more fully, please see the usage section below.

Helper Initialization

A helper can be initialized in several different ways, based on your needs as well as the functionality of that helper.

The helper broker is stored as the *$_helper* member of Zend_Controller_Action; use the broker to retrieve or call on helpers. Some methods for doing so include:

- Explicitly using getHelper(). Simply pass it a name, and a helper object is returned:

```
1. $flashMessenger = $this->_helper->getHelper('FlashMessenger');
2. $flashMessenger->addMessage('We did something in the last request');
```

- Use the helper broker's __get() functionality and retrieve the helper as if it were a member property of the broker:

```
1. $flashMessenger = $this->_helper->FlashMessenger;
2. $flashMessenger->addMessage('We did something in the last request');
```

- Finally, most action helpers implement the method direct() which will call a specific, default method in the helper. In the example of the *FlashMessenger*, it calls addMessage():

```
1. $this->_helper->FlashMessenger('We did something in the last request');
```

Note: All of the above examples are functionally equivalent.

You may also instantiate helpers explicitly. You may wish to do this if using the helper outside of an action controller, or if you wish to pass a helper to the helper broker for use by any action. Instantiation is as per any other PHP class.

The Helper Broker

Zend_Controller_Action_HelperBroker handles the details of registering helper objects and helper paths, as well as retrieving helpers on-demand.

To register a helper with the broker, use addHelper():

```
1. Zend_Controller_Action_HelperBroker::addHelper($helper);
```

Of course, instantiating and passing helpers to the broker is a bit time and resource intensive, so two methods exists to automate things slightly: addPrefix() and addPath().

- addPrefix() takes a class prefix and uses it to determine a path where helper classes have been defined. It assumes the prefix follows Zend Framework class naming conventions.

```
1. // Add helpers prefixed with My_Action_Helpers in My/Action/Helpers/
2. Zend_Controller_Action_HelperBroker::addPrefix('My_Action_Helpers');
```

- addPath() takes a directory as its first argument and a class prefix as the second argument (defaulting to 'Zend_Controller_Action_Helper'). This allows you to map your own class prefixes to specific directories.

```
1. // Add helpers prefixed with Helper in Plugins/Helpers/
2. Zend_Controller_Action_HelperBroker::addPath('./Plugins/Helpers',
3.                                              'Helper');
```

Since these methods are static, they may be called at any point in the controller chain in order to dynamically add helpers as needed.

Internally, the helper broker uses a PluginLoader instance to maintain paths. You can retrieve the PluginLoader using the static method getPluginLoader(), or, alternately, inject a custom PluginLoader instance using setPluginLoader().

To determine if a helper exists in the helper broker, use hasHelper($name), where *$name* is the short name of the helper (minus the prefix):

```
1. // Check if 'redirector' helper is registered with the broker:
2. if (Zend_Controller_Action_HelperBroker::hasHelper('redirector')) {
3.     echo 'Redirector helper registered';
4. }
```

There are also two static methods for retrieving helpers from the helper broker: getExistingHelper() and getStaticHelper(). getExistingHelper() will retrieve a helper only if it has previously been invoked by or explicitly registered with the helper broker; it will throw an exception if not. getStaticHelper() does the same as getExistingHelper(), but will attempt to instantiate the helper if has not yet been registered with the helper stack. getStaticHelper() is a good choice for retrieving helpers which you wish to configure.

Both methods take a single argument, *$name*, which is the short name of the helper (minus the prefix).

```
1. // Check if 'redirector' helper is registered with the broker, and fetch:
2. if (Zend_Controller_Action_HelperBroker::hasHelper('redirector')) {
```

```
    $redirector =
Zend_Controller_Action_HelperBroker::getExistingHelper('redirector');
}

// Or, simply retrieve it, not worrying about whether or not it was
// previously registered:
$redirector =
    Zend_Controller_Action_HelperBroker::getStaticHelper('redirector');
}
```

Finally, to delete a registered helper from the broker, use removeHelper($name), where *$name* is the short name of the helper (minus the prefix):

```
// Conditionally remove the 'redirector' helper from the broker:
if (Zend_Controller_Action_HelperBroker::hasHelper('redirector')) {
    Zend_Controller_Action_HelperBroker::removeHelper('redirector')
}
```

Built-in Action Helpers

Zend Framework includes several action helpers by default: *AutoComplete* for automating responses for AJAX autocompletion; *ContextSwitch* and *AjaxContext* for serving alternate response formats for your actions; a *FlashMessenger* for handling session flash messages; *Json* for encoding and sending JSON responses; a *Redirector*, to provide different implementations for redirecting to internal and external pages from your application; and a *ViewRenderer* to automate the process of setting up the view object in your controllers and rendering views.

ActionStack

The *ActionStack* helper allows you to push requests to the ActionStack front controller plugin, effectively helping you create a queue of actions to execute during the request. The helper allows you to add actions either by specifying new request objects or action - controller - module sets.

> **Note**: **Invoking ActionStack Helper Initializes the ActionStack Plugin**
> Invoking the *ActionStack* helper implicitly registers the *ActionStack* plugin -- which means you do not need to explicitly register the *ActionStack* plugin to use this functionality.

Example #1 Adding a Task Using Action, Controller and Module Names

Often, it's simplest to simply specify the action, controller, and module (and optional request parameters), much as you would when calling Zend_Controller_Action::_forward():

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // Add two actions to the stack
        // Add call to /foo/baz/bar/baz
        // (FooController::bazAction() with request var bar == baz)
        $this->_helper->actionStack('baz',
                                    'foo',
                                    'default',
                                    array('bar' => 'baz'));

        // Add call to /bar/bat
        // (BarController::batAction())
        $this->_helper->actionStack('bat', 'bar');
```

```
    }
}
```

Example #2 Adding a Task Using a Request Object

Sometimes the OOP nature of a request object makes most sense; you can pass such an object to the *ActionStack* helper as well.

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // Add two actions to the stack
        // Add call to /foo/baz/bar/baz
        // (FooController::bazAction() with request var bar == baz)
        $request = clone $this->getRequest();
        // Don't set controller or module; use current values
        $request->setActionName('baz')
                ->setParams(array('bar' => 'baz'));
        $this->_helper->actionStack($request);

        // Add call to /bar/bat
        // (BarController::batAction())
        $request = clone $this->getRequest();
        // don't set module; use current value
        $request->setActionName('bat')
                ->setControllerName('bar');
        $this->_helper->actionStack($request);
    }
}
```

AutoComplete

Many AJAX javascript libraries offer functionality for providing autocompletion whereby a selectlist of potentially matching results is displayed as the user types. The *AutoComplete* helper aims to simplify returning acceptable responses to such methods.

Since not all JS libraries implement autocompletion in the same way, the *AutoComplete* helper provides some abstract base functionality necessary to many libraries, and concrete implementations for individual libraries. Return types are generally either JSON arrays of strings, JSON arrays of arrays (with each member array being an associative array of metadata used to create the selectlist), or HTML.

Basic usage for each implementation is the same:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // Perform some logic...

        // Encode and send response;
        $this->_helper->autoCompleteDojo($data);

        // Or explicitly:
        $response = $this->_helper->autoCompleteDojo
                                  ->sendAutoCompletion($data);

        // Or simply prepare autocompletion response:
```

```
15.         $response = $this->_helper->autoCompleteDojo
16.                                   ->prepareAutoCompletion($data);
17.     }
18. }
```

By default, autocompletion does the following:

- Disables layouts and ViewRenderer.
- Sets appropriate response headers.
- Sets response body with encoded or formatted autocompletion data.
- Sends response.

Available methods of the helper include:

- disableLayouts() can be used to disable layouts and the ViewRenderer. Typically, this is called within prepareAutoCompletion().
- encodeJson($data, $keepLayouts = false) will encode data to JSON, optionally enabling or disabling layouts. Typically, this is called within prepareAutoCompletion().
- prepareAutoCompletion($data, $keepLayouts = false) is used to prepare data in the response format necessary for the concrete implementation, optionally enabling or disabling layouts. The return value will vary based on the implementation.
- sendAutoCompletion($data, $keepLayouts = false) is used to send data in the response format necessary for the concrete implementation. It calls prepareAutoCompletion(), and then sends the response.
- direct($data, $sendNow = true, $keepLayouts = false) is used when calling the helper as a method of the helper broker. The *$sendNow* flag is used to determine whether to call sendAutoCompletion() or prepareAutoCompletion(), respectively.

Currently, *AutoComplete* supports the Dojo and Scriptaculous AJAX libraries.

AutoCompletion with Dojo

Dojo does not have an AutoCompletion widget per se, but has two widgets that can perform AutoCompletion: ComboBox and FilteringSelect. In both cases, they require a data store that implements the QueryReadStore; for more information on these topics, see the » dojo.data documentation.

In Zend Framework, you can pass a simple indexed array to the AutoCompleteDojo helper, and it will return a JSON response suitable for use with such a store:

```
1. // within a controller action:
2. $this->_helper->autoCompleteDojo($data);
```

Example #3 AutoCompletion with Dojo Using Zend MVC

AutoCompletion with Dojo via the Zend MVC requires several things: generating a form object for the ComboBox on which you want AutoCompletion, a controller action for serving the AutoCompletion results, creating a custom QueryReadStore to connect to the AutoCompletion action, and generation of the javascript to use to initialize AutoCompletion on the server side.

First, let's look at the javascript necessary. Dojo offers a complete framework for creating OOP javascript, much as Zend Framework does for PHP. Part of that is the ability to create pseudo-namespaces using the directory hierarchy. We'll create a 'custom' directory at the same level as the Dojo directory that's part of the Dojo distribution. Inside that directory, we'll create a javascript file, *TestNameReadStore.js*, with the following contents:

```
dojo.provide("custom.TestNameReadStore");
dojo.declare("custom.TestNameReadStore", dojox.data.QueryReadStore, {
    fetch:function (request) {
        request.serverQuery = { test:request.query.name };
        return this.inherited("fetch", arguments);
    }
});
```

This class is simply an extension of Dojo's own QueryReadStore, which is itself an abstract class. We simply define a method by which to request, and assigning it to the 'test' element.

Next, let's create the form element for which we want AutoCompletion:

```
class TestController extends Zend_Controller_Action
{
    protected $_form;

    public function getForm()
    {
        if (null === $this->_form) {
            $this->_form = new Zend_Form();
            $this->_form->setMethod('get')
                ->setAction(
                    $this->getRequest()->getBaseUrl() . '/test/process'
                )
                ->addElements(array(
                    'test' => array('type' => 'text', 'options' => array(
                        'filters'        => array('StringTrim'),
                        'dojoType'       => array('dijit.form.ComboBox'),
                        'store'          => 'testStore',
                        'autoComplete'   => 'false',
                        'hasDownArrow'   => 'true',
                        'label' => 'Your input:',
                    )),
                    'go' => array('type' => 'submit',
                                  'options' => array('label' => 'Go!'))
                ));
        }
        return $this->_form;
    }
}
```

Here, we simply create a form with 'test' and 'go' methods. The 'test' method adds several special, Dojo-specific attributes: dojoType, store, autoComplete, and hasDownArrow. The dojoType is used to indicate that we are creating a ComboBox, and we will link it to a data store (key 'store') of 'testStore' -- more on that later. Specifying 'autoComplete' as **FALSE** tells Dojo not to automatically select the first match, but instead show a list of matches. Finally, 'hasDownArrow' creates a down arrow similar to a select box so we can show and hide the matches.

Let's add a method to display the form, as well as an end point for processing AutoCompletion:

```
class TestController extends Zend_Controller_Action
{
    // ...

    /**
     * Landing page
     */
    public function indexAction()
```

```
    {
        $this->view->form = $this->getForm();
    }

    public function autocompleteAction()
    {
        if ('ajax' != $this->_getParam('format', false)) {
            return $this->_helper->redirector('index');
        }
        if ($this->getRequest()->isPost()) {
            return $this->_helper->redirector('index');
        }

        $match = trim($this->getRequest()->getQuery('test', ''));

        $matches = array();
        foreach ($this->getData() as $datum) {
            if (0 === strpos($datum, $match)) {
                $matches[] = $datum;
            }
        }
        $this->_helper->autoCompleteDojo($matches);
    }
}
```

In our autocompleteAction() we do a number of things. First, we look to make sure we have a post request, and that there is a 'format' parameter set to the value 'ajax'; these are simply to help reduce spurious queries to the action. Next, we check for a 'test' parameter, and compare it against our data. (I purposely leave out the implementation of getData() here -- it could be any sort of data source.) Finally, we send our matches to our AutoCompletion helper.

Now that we have all the pieces on the backend, let's look at what we need to deliver in our view script for the landing page. First, we need to setup our data store, then render our form, and finally ensure that the appropriate Dojo libraries -- including our custom data store -- get loaded. Let's look at the view script, which comments the steps:

```
<?php // setup our data store: ?>
<div dojoType="custom.TestNameReadStore" jsId="testStore"
    url="<?php echo $this->baseUrl() ?>/unit-test/autocomplete/format/ajax"
    requestMethod="get"></div>

<?php // render our form: ?>
<?php echo $this->form ?>

<?php // setup Dojo-related CSS to load in HTML head: ?>
<?php $this->headStyle()->captureStart() ?>
@import "<?php echo $this->baseUrl()
?>/javascript/dijit/themes/tundra/tundra.css";
@import "<?php echo $this->baseUrl() ?>/javascript/dojo/resources/dojo.css";
<?php $this->headStyle()->captureEnd() ?>

<?php // setup javascript to load in HTML head, including all required
   // Dojo libraries: ?>
<?php $this->headScript()
        ->setAllowArbitraryAttributes(true)
        ->appendFile($this->baseUrl() . '/javascript/dojo/dojo.js',
            'text/javascript',
            array('djConfig' => 'parseOnLoad: true'))
        ->captureStart() ?>
djConfig.usePlainJson=true;
```

```
dojo.registerModulePath("custom","../custom");
dojo.require("dojo.parser");
dojo.require("dojox.data.QueryReadStore");
dojo.require("dijit.form.ComboBox");
dojo.require("custom.TestNameReadStore");
<?php $this->headScript()->captureEnd() ?>
```

Note the calls to view helpers such as headStyle and headScript; these are placeholders, which we can then render in the HTML head section of our layout view script.

We now have all the pieces to get Dojo AutoCompletion working.

AutoCompletion with Scriptaculous

» Scriptaculous expects an HTML response in a specific format.

The helper to use with this library is 'AutoCompleteScriptaculous'. Simply provide it an array of data, and the helper will create an HTML response compatible with Ajax.Autocompleter.

ContextSwitch and AjaxContext

The *ContextSwitch* action helper is intended for facilitating returning different response formats on request. The *AjaxContext* helper is a specialized version of *ContextSwitch* that facilitates returning responses to XmlHttpRequests.

To enable either one, you must provide hinting in your controller as to what actions can respond to which contexts. If an incoming request indicates a valid context for the given action, the helper will then:

- Disable layouts, if enabled.
- Set an alternate view suffix, effectively requiring a separate view script for the context.
- Send appropriate response headers for the context desired.
- Optionally, call specified callbacks to setup the context and/or perform post-processing.

As an example, let's consider the following controller:

```
class NewsController extends Zend_Controller_Action
{
    /**
     * Landing page; forwards to listAction()
     */
    public function indexAction()
    {
        $this->_forward('list');
    }

    /**
     * List news items
     */
    public function listAction()
    {
    }

    /**
     * View a news item
     */
    public function viewAction()
```

```
    {
    }
}
```

Let's say that we want the listAction() to also be available in an XML format. Instead of creating a different action, we can hint that it can return an XML response:

```
class NewsController extends Zend_Controller_Action
{
    public function init()
    {
        $contextSwitch = $this->_helper->getHelper('contextSwitch');
        $contextSwitch->addActionContext('list', 'xml')
                      ->initContext();
    }

    // ...
}
```

What this will do is:

- Set the 'Content-Type' response header to '*application/xml*'.
- Change the view suffix to '*xml.phtml*' (or, if you use an alternate view suffix, 'xml.[your suffix]').

Now, you'll need to create a new view script, '*news/list.xml.phtml*', which will create and render the XML.

To determine if a request should initiate a context switch, the helper checks for a token in the request object. By default, it looks for the 'format' parameter, though this may be configured. This means that, in most cases, to trigger a context switch, you can add a 'format' parameter to your request:

- Via URL parameter: */news/list/format/xml* (recall, the default routing schema allows for arbitrary key to value pairs following the action)
- Via **`GET`** parameter: **/news/list?format=xml**

ContextSwitch allows you to specify arbitrary contexts, including what suffix change will occur (if any), any response headers that should be sent, and arbitrary callbacks for initialization and post processing.

Default Contexts Available

By default, two contexts are available to the *ContextSwitch* helper: json and XML.

- *JSON*. The JSON context sets the 'Content-Type' response header to '*application/json*', and the view script suffix to '*json.phtml*'.

  By default, however, no view script is required. It will simply serialize all view variables, and emit the JSON response immediately.

  This behaviour can be disabled by turning off the automatic JSON serialization:

  ```
  $this->_helper->contextSwitch()->setAutoJsonSerialization(false);
  ```

- *XML*. The XML context sets the 'Content-Type' response header to '*application/xml*', and the view script suffix to '*xml.phtml*'. You will need to create a new view script for the context.

Creating Custom Contexts

Sometimes, the default contexts are not enough. For instance, you may wish to return YAML, or serialized PHP, an RSS

or ATOM feed, etc. *ContextSwitch* allows you to do so.

The easiest way to add a new context is via the addContext() method. This method takes two arguments, the name of the context, and an array specification. The specification should include one or more of the following:

- *suffix*: the suffix to prepend to the default view suffix as registered in the ViewRenderer.
- *headers*: an array of header to value pairs you wish sent as part of the response.
- *callbacks*: an array containing one or more of the keys 'init' or 'post', pointing to valid PHP callbacks that can be used for context initialization and post processing.

  Initialization callbacks occur when the context is detected by *ContextSwitch*. You can use it to perform arbitrary logic that should occur. As an example, the JSON context uses a callback to disable the ViewRenderer when the automatic JSON serialization is on.

  Post processing occurs during the action's postDispatch() routine, and can be used to perform arbitrary logic. As an example, the JSON context uses a callback to determine if the automatic JSON serialization is on; if so, it serializes the view variables to JSON and sends the response, but if not, it re-enables the ViewRenderer.

There are a variety of methods for interacting with contexts:

- addContext($context, array $spec): add a new context. Throws an exception if the context already exists.
- setContext($context, array $spec): add a new context or overwrite an existing context. Uses the same specification as addContext().
- addContexts(array $contexts): add many contexts at once. The *$contexts* array should be an array of context to specification pairs. If any of the contexts already exists, it will throw an exception.
- setContexts(array $contexts): add new contexts and overwrite existing ones. Uses the same specification as addContexts().
- hasContext($context): returns **TRUE** if the context exists, **FALSE** otherwise.
- getContext($context): retrieve a single context by name. Returns an array following the specification used in addContext().
- getContexts(): retrieve all contexts. Returns an array of context to specification pairs.
- removeContext($context): remove a single context by name. Returns **TRUE** if successful, **FALSE** if the context was not found.
- clearContexts(): remove all contexts.

Setting Contexts Per Action

There are two mechanisms for setting available contexts. You can either manually create arrays in your controller, or use several methods in *ContextSwitch* to assemble them.

The principle method for adding action to context relations is addActionContext(). It expects two arguments, the action to which the context is being added, and either the name of a context or an array of contexts. As an example, consider the following controller class:

```
1.  class FooController extends Zend_Controller_Action
2.  {
3.      public function listAction()
4.      {
5.      }
```

```
6.
7.      public function viewAction()
8.      {
9.      }
10.
11.     public function commentsAction()
12.     {
13.     }
14.
15.     public function updateAction()
16.     {
17.     }
18. }
```

Let's say we wanted to add an XML context to the 'list' action, and XML and JSON contexts to the 'comments' action. We could do so in the init() method:

```
1.  class FooController extends Zend_Controller_Action
2.  {
3.      public function init()
4.      {
5.          $this->_helper->contextSwitch()
6.               ->addActionContext('list', 'xml')
7.               ->addActionContext('comments', array('xml', 'json'))
8.               ->initContext();
9.      }
10. }
```

Alternately, you could simply define the array property *$contexts*:

```
1.  class FooController extends Zend_Controller_Action
2.  {
3.      public $contexts = array(
4.          'list'     => array('xml'),
5.          'comments' => array('xml', 'json')
6.      );
7.
8.      public function init()
9.      {
10.         $this->_helper->contextSwitch()->initContext();
11.     }
12. }
```

The above is less overhead, but also prone to potential errors.

The following methods can be used to build the context mappings:

- addActionContext($action, $context): marks one or more contexts as available to an action. If mappings already exists, simply appends to those mappings. *$context* may be a single context, or an array of contexts.

  A value of **TRUE** for the context will mark all available contexts as available for the action.

  An empty value for *$context* will disable all contexts for the given action.

- setActionContext($action, $context): marks one or more contexts as available to an action. If mappings already exists, it replaces them with those specified. *$context* may be a single context, or an array of contexts.

- addActionContexts(array $contexts): add several action to context pairings at once. *$contexts* should be an associative array of action to context pairs. It proxies to addActionContext(), meaning that if pairings already

exist, it appends to them.

- setActionContexts(array $contexts): acts like addActionContexts(), but overwrites existing action to context pairs.
- hasActionContext($action, $context): determine if a particular action has a given context.
- getActionContexts($action = null): returns either all contexts for a given action, or all action to context pairs.
- removeActionContext($action, $context): remove one or more contexts from a given action. *$context* may be a single context or an array of contexts.
- clearActionContexts($action = null): remove all contexts from a given action, or from all actions with contexts.

Initializing Context Switching

To initialize context switching, you need to call initContext() in your action controller:

```
1.  class NewsController extends Zend_Controller_Action
2.  {
3.      public function init()
4.      {
5.          $this->_helper->contextSwitch()->initContext();
6.      }
7.  }
```

In some cases, you may want to force the context used; for instance, you may only want to allow the XML context if context switching is activated. You can do so by passing the context to initContext():

```
1.  $contextSwitch->initContext('xml');
```

Additional Functionality

A variety of methods can be used to alter the behaviour of the *ContextSwitch* helper. These include:

- setAutoJsonSerialization($flag): By default, JSON contexts will serialize any view variables to JSON notation and return this as a response. If you wish to create your own response, you should turn this off; this needs to be done prior to the call to initContext().

  ```
  1.  $contextSwitch->setAutoJsonSerialization(false);
  2.  $contextSwitch->initContext();
  ```

  You can retrieve the value of the flag with getAutoJsonSerialization().

- setSuffix($context, $suffix, $prependViewRendererSuffix): With this method, you can specify a different suffix to use for a given context. The third argument is used to indicate whether or not to prepend the current ViewRenderer suffix with the new suffix; this flag is enabled by default.

  Passing an empty value to the suffix will cause only the ViewRenderer suffix to be used.

- addHeader($context, $header, $content): Add a response header for a given context. *$header* is the header name, and *$content* is the value to pass for that header.

  Each context can have multiple headers; addHeader() adds additional headers to the context's header stack.

  If the *$header* specified already exists for the context, an exception will be thrown.

- setHeader($context, $header, $content): setHeader() acts just like addHeader(), except it allows you to overwrite existing context headers.

- addHeaders($context, array $headers): Add multiple headers at once to a given context. Proxies to addHeader(), so if the header already exists, an exception will be thrown. *$headers* is an array of header to context pairs.
- setHeaders($context, array $headers.): like addHeaders(), except it proxies to setHeader(), allowing you to overwrite existing headers.
- getHeader($context, $header): retrieve the value of a header for a given context. Returns `NULL` if not found.
- removeHeader($context, $header): remove a single header for a given context.
- clearHeaders($context, $header): remove all headers for a given context.
- setCallback($context, $trigger, $callback): set a callback at a given trigger for a given context. Triggers may be either 'init' or 'post' (indicating callback will be called at either context initialization or postDispatch). *$callback* should be a valid PHP callback.
- setCallbacks($context, array $callbacks): set multiple callbacks for a given context. *$callbacks* should be trigger to callback pairs. In actuality, the most callbacks that can be registered are two, one for initialization and one for post processing.
- getCallback($context, $trigger): retrieve a callback for a given trigger in a given context.
- getCallbacks($context): retrieve all callbacks for a given context. Returns an array of trigger to callback pairs.
- removeCallback($context, $trigger): remove a callback for a given trigger and context.
- clearCallbacks($context): remove all callbacks for a given context.
- setContextParam($name): set the request parameter to check when determining if a context switch has been requested. The value defaults to 'format', but this accessor can be used to set an alternate value.

  getContextParam() can be used to retrieve the current value.
- setAutoDisableLayout($flag): By default, layouts are disabled when a context switch occurs; this is because typically layouts will only be used for returning normal responses, and have no meaning in alternate contexts. However, if you wish to use layouts (perhaps you may have a layout for the new context), you can change this behaviour by passing a `FALSE` value to setAutoDisableLayout(). You should do this *before* calling initContext().

  To get the value of this flag, use the accessor getAutoDisableLayout().
- getCurrentContext() can be used to determine what context was detected, if any. This returns `NULL` if no context switch occurred, or if called before initContext() has been invoked.

AjaxContext Functionality

The *AjaxContext* helper extends *ContextSwitch*, so all of the functionality listed for *ContextSwitch* is available to it. There are a few key differences, however.

First, it uses a different action controller property for determining contexts, *$ajaxable*. This is so you can have different contexts used for AJAX versus normal HTTP requests. The various * ActionContext()* methods of *AjaxContext* will write to this property.

Second, it will only trigger if an XmlHttpRequest has occurred, as determined by the request object's isXmlHttpRequest() method. Thus, if the context parameter ('format') is passed in the request, but the request was not made as an XmlHttpRequest, no context switch will trigger.

Third, *AjaxContext* adds an additional context, HTML. In this context, it sets the suffix to '*ajax.phtml*' in order to differentiate the context from a normal request. No additional headers are returned.

Example #4 Allowing Actions to Respond To Ajax Requests

In this following example, we're allowing requests to the actions 'view', 'form', and 'process' to respond to AJAX requests. In the first two cases, 'view' and 'form', we'll return HTML snippets with which to update the page; in the latter, we'll return JSON.

```
class CommentController extends Zend_Controller_Action
{
    public function init()
    {
        $ajaxContext = $this->_helper->getHelper('AjaxContext');
        $ajaxContext->addActionContext('view', 'html')
                    ->addActionContext('form', 'html')
                    ->addActionContext('process', 'json')
                    ->initContext();
    }

    public function viewAction()
    {
        // Pull a single comment to view.
        // When AjaxContext detected, uses the comment/view.ajax.phtml
        // view script.
    }

    public function formAction()
    {
        // Render the "add new comment" form.
        // When AjaxContext detected, uses the comment/form.ajax.phtml
        // view script.
    }

    public function processAction()
    {
        // Process a new comment
        // Return the results as JSON; simply assign the results as
        // view variables, and JSON will be returned.
    }
}
```

On the client end, your AJAX library will simply request the endpoints '*/comment/view*', '*/comment/form*', and '*/comment/process*', and pass the 'format' parameter: '*/comment/view/format/html*', '*/comment/form/format/html*', '*/comment/process/format/json*'. (Or you can pass the parameter via query string: e.g., "?format=json".)

Assuming your library passes the 'X-Requested-With: XmlHttpRequest' header, these actions will then return the appropriate response format.

FlashMessenger

Introduction

The *FlashMessenger* helper allows you to pass messages that the user may need to see on the next request. To accomplish this, *FlashMessenger* uses Zend_Session_Namespace to store messages for future or next request retrieval. It is generally a good idea that if you plan on using Zend_Session or Zend_Session_Namespace, that you initialize with Zend_Session::start() in your bootstrap file. (See the Zend_Session documentation for more details on its usage.)

Available Methods

General methods:

- setNamespace($namespace='default') is used to set the namespace into which messages are stored by default.
- getNamespace() is used to retrieve the name of the default namespace. The default namespace is 'default'.
- resetNamespace() is used to reset the namespace name to the default value, 'default'.

Methods for manipulating messages set in the previous request:

- hasMessages($namespace=NULL) is used to determine if messages have been carried from a previous request by the flash messenger. The optional argument *$namespace* specifies which namespace to look in. If the *$namespace* argument is omitted, the value returned by getNamespace() will be used.
- getMessages($namespace=NULL) is used to retrieve the messages which have been carried from a previous request by the flash messenger. The optional argument *$namespace* specifies which namespace to pull from. If the *$namespace* argument is omitted, the value returned by getNamespace() will be used.
- getIterator($namespace=NULL) wraps the return value of getMessages() in an instance of ArrayObject. If the *$namespace* argument is omitted, the value returned by getNamespace() will be used.
- count($namespace=NULL) returns the number of messages contained in the specified namespace. If the *$namespace* argument is omitted, the value returned by getNamespace() will be used.
- clearMessages($namespace=NULL) is used to clear all the messages which have been carried from a previous request by the flash messenger. The optional argument *$namespace* specifies which namespace to clear out. If the *$namespace* argument is omitted, the value returned by getNamespace() will be used.

Methods for manipulating messages set in the current request:

- addMessage($message, $namespace=NULL) is used to add a new message to the current request. *$message* contains the message to be added, and the optional argument *$namespace* will specify the namespace. If the *$namespace* argument is omitted, the value returned by getNamespace() will be used.
- hasCurrentMessages($namespace=NULL) is used to determine if messages have been added to the flash messenger during the current request. The optional argument *$namespace* specifies which namespace to look in. If the *$namespace* argument is omitted, the value returned by getNamespace() will be used.
- getCurrentMessages($namespace=NULL) is used to retrieve the messages which have been added to the flash messenger during the current request. The optional argument *$namespace* specifies which namespace to pull from. If the *$namespace* argument is omitted, the value returned by getNamespace() will be used.
- clearCurrentMessages($namespace=NULL) is used to clear all the messages which have been added to the flash messenger during the current request. The optional argument *$namespace* specifies which namespace to clear out. If the *$namespace* argument is omitted, the value returned by getNamespace() will be used.

Basic Usage Example

The usage example below shows the use of the flash messenger at its most basic. When the action */some/my* is called, it adds the flash message "Record Saved!" A subsequent request to the action */some/my-next-request* will retrieve it (and thus delete it as well).

```
1. class SomeController extends Zend_Controller_Action
2. {
3.     /**
4.      * FlashMessenger
5.      *
6.      * @var Zend_Controller_Action_Helper_FlashMessenger
7.      */
```

```
 8.      protected $_flashMessenger = null;
 9.
10.      public function init()
11.      {
12.          $this->_flashMessenger =
13.              $this->_helper->getHelper('FlashMessenger');
14.          $this->initView();
15.      }
16.
17.      public function myAction()
18.      {
19.          /**
20.           * default method of getting
21.           * Zend_Controller_Action_Helper_FlashMessenger instance
22.           * on-demand
23.           */
24.          $this->_flashMessenger->addMessage('Record Saved!');
25.      }
26.
27.      public function myNextRequestAction()
28.      {
29.          $this->view->messages = $this->_flashMessenger->getMessages();
30.          $this->render();
31.      }
32.  }
```

JSON

JSON responses are rapidly becoming the response of choice when dealing with AJAX requests that expect dataset responses; JSON can be immediately parsed on the client-side, leading to quick execution.

Usage

Usage is simple: either call it as a method of the helper broker, or call one of the methods encodeJson() or sendJson():

direct($data, $sendNow = true, $keepLayouts = false, $encodeData = true)

sendJson($data, $keepLayouts = false, $encodeData = true)

encodeJson($data, $keepLayouts = false, $encodeData = true)

- *$data*: data to encode as JSON
- *$sendNow*: flag to define whether to send the JSON data immediately. When true, the helper will immediately set the respose body and exit.
- *$keepLayouts*: flag to define whether to enable or disable layours. When false, all layouts are disabled. Optionally, this can be an array of options to pass as the second argument to Zend_Json::encode(). This array of options allows enabling layouts and encoding using Zend_Json_Expr.
- *$encodeData*: flag to define whether *$data* is already JSON-encoded. When true, this helper will not encode *$data* to JSON before sending.

  Note: **Keeping Layouts**
  If you have a separate layout for JSON responses -- perhaps to wrap the JSON response in some sort of context -- each method in the JSON helper accepts an optional argument *$keepLayouts*: a flag to enable or disable layouts. Passing a boolean **TRUE** value will keep layouts enabled:

```
$this->_helper->json($data, true);
```

Optionally, you can pass an array as the third parameter. This array may contain a variety of options, including the *keepLayouts* option:

```
// Direct helper call
$this->_helper->json($data, true, array('keepLayouts' => true);

// ...or, call a method of the helper
$this->_helper->sendJson($data, array('keepLayouts' => true));
```

Note: **Enabling encoding using Zend_Json_Expr**
Zend_Json::encode() allows the encoding of native JSON expressions using Zend_Json_Expr objects. This option is disabled by default. To enable this option, pass a boolean **TRUE** value to the *enableJsonExprFinder* option:

```
$this->_helper->json($data, true, array('enableJsonExprFinder' =>
true);
```

If you desire to do this, you *must* pass an array as the third argument. This also allows you to combine other options, such as the *keepLayouts* option. All such options are then passed to Zend_Json::encode().

```
$this->_helper->json($data, true, array(
'enableJsonExprFinder' => true,
'keepLayouts'          => true,
));
```

Example

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // do some processing...
        // Send the JSON response:
        $this->_helper->json($data);

        // or...
        $this->_helper->json->sendJson($data);

        // or retrieve the json:
        $json = $this->_helper->json->encodeJson($data);
    }
}
```

Redirector

Introduction

The *Redirector* helper allows you to use a redirector object to fulfill your application's needs for redirecting to a new URL. It provides numerous benefits over the _redirect() method, such as being able to preconfigure sitewide behavior into the redirector object or using the built in gotoSimple($action, $controller, $module, $params) interface similar to that of Zend_Controller_Action::_forward().

The *Redirector* has a number of methods that can be used to affect the behaviour at redirect:

- setCode() can be used to set the HTTP response code to use during the redirect.

- setExit() can be used to force an exit() following a redirect. By default this is **TRUE**.
- setGotoSimple() can be used to set a default URL to use if none is passed to gotoSimple(). Uses the API of Zend_Controller_Action::_forward(): setGotoSimple($action, $controller = null, $module = null, array $params = array())
- setGotoRoute() can be used to set a URL based on a registered route. Pass in an array of key / value pairs and a route name, and it will assemble the URL according to the route type and definition.
- setGotoUrl() can be used to set a default URL to use if none is passed to gotoUrl(). Accepts a single URL string.
- setPrependBase() can be used to prepend the request object's base URL to a URL specified with setGotoUrl(), gotoUrl(), or gotoUrlAndExit().
- setUseAbsoluteUri() can be used to force the *Redirector* to use absolute URIs when redirecting. When this option is set, it uses the value of *$_SERVER['HTTP_HOST']*, *$_SERVER['SERVER_PORT']*, and *$_SERVER['HTTPS']* to form a full URI to the URL specified by one of the redirect methods. This option is off by default, but may be enabled by default in later releases.

Additionally, there are a variety of methods in the redirector for performing the actual redirects:

- gotoSimple() uses setGotoSimple() (_forward()-like API) to build a URL and perform a redirect.
- gotoRoute() uses setGotoRoute() (*route-assembly*) to build a URL and perform a redirect.
- gotoUrl() uses setGotoUrl() (*URL string*) to build a URL and perform a redirect.

Finally, you can determine the current redirect URL at any time using getRedirectUrl().

Basic Usage Examples

Example #5 Setting Options

This example overrides several options, including setting the HTTP status code to use in the redirect ('303'), not defaulting to exit on redirect, and defining a default URL to use when redirecting.

```
class SomeController extends Zend_Controller_Action
{
    /**
     * Redirector - defined for code completion
     *
     * @var Zend_Controller_Action_Helper_Redirector
     */
    protected $_redirector = null;

    public function init()
    {
        $this->_redirector = $this->_helper->getHelper('Redirector');

        // Set the default options for the redirector
        // Since the object is registered in the helper broker, these
        // become relevant for all actions from this point forward
        $this->_redirector->setCode(303)
                          ->setExit(false)
                          ->setGotoSimple("this-action",
                                          "some-controller");
    }

    public function myAction()
```

```
    {
        /* do some stuff */

        // Redirect to a previously registered URL, and force an exit
        // to occur when done:
        $this->_redirector->redirectAndExit();
        return; // never reached
    }
}
```

Example #6 Using Defaults

This example assumes that the defaults are used, which means that any redirect will result in an immediate exit().

```
// ALTERNATIVE EXAMPLE
class AlternativeController extends Zend_Controller_Action
{
    /**
     * Redirector - defined for code completion
     *
     * @var Zend_Controller_Action_Helper_Redirector
     */
    protected $_redirector = null;

    public function init()
    {
        $this->_redirector = $this->_helper->getHelper('Redirector');
    }

    public function myAction()
    {
        /* do some stuff */

        $this->_redirector
            ->gotoUrl('/my-controller/my-action/param1/test/param2/test2');
        return; // never reached since default is to goto and exit
    }
}
```

Example #7 Using goto()'s _forward() API

gotoSimple()'s API mimics that of Zend_Controller_Action::_forward(). The primary difference is that it builds a URL from the parameters passed, and using the default *:module/:controller/:action/** format of the default router. It then redirects instead of chaining the action.

```
class ForwardController extends Zend_Controller_Action
{
    /**
     * Redirector - defined for code completion
     *
     * @var Zend_Controller_Action_Helper_Redirector
     */
    protected $_redirector = null;

    public function init()
    {
        $this->_redirector = $this->_helper->getHelper('Redirector');
    }
```

```
14.
15.     public function myAction()
16.     {
17.         /* do some stuff */
18.
19.         // Redirect to 'my-action' of 'my-controller' in the current
20.         // module, using the params param1 => test and param2 => test2
21.         $this->_redirector->gotoSimple('my-action',
22.                                        'my-controller',
23.                                        null,
24.                                        array('param1' => 'test',
25.                                              'param2' => 'test2'
26.                                              )
27.                                        );
28.     }
29. }
```

Example #8 Using Route Assembly with gotoRoute()

The following example uses the router's assemble() method to create a URL based on an associative array of parameters passed. It assumes the following route has been registered:

```
1.  $route = new Zend_Controller_Router_Route(
2.      'blog/:year/:month/:day/:id',
3.      array('controller' => 'archive',
4.            'module' => 'blog',
5.            'action' => 'view')
6.  );
7.  $router->addRoute('blogArchive', $route);
```

Given an array with year set to 2006, month to 4, day to 24, and id to 42, it would then build the URL */blog/2006/4/24/42*.

```
1.  class BlogAdminController extends Zend_Controller_Action
2.  {
3.      /**
4.       * Redirector - defined for code completion
5.       *
6.       * @var Zend_Controller_Action_Helper_Redirector
7.       */
8.      protected $_redirector = null;
9.
10.     public function init()
11.     {
12.         $this->_redirector = $this->_helper->getHelper('Redirector');
13.     }
14.
15.     public function returnAction()
16.     {
17.         /* do some stuff */
18.
19.         // Redirect to blog archive. Builds the following URL:
20.         // /blog/2006/4/24/42
21.         $this->_redirector->gotoRoute(
22.             array('year' => 2006,
23.                   'month' => 4,
24.                   'day' => 24,
25.                   'id' => 42),
```

```
            'blogArchive'
        );
    }
}
```

ViewRenderer

Introduction

The *ViewRenderer* helper is designed to satisfy the following goals:

- Eliminate the need to instantiate view objects within controllers; view objects will be automatically registered with the controller.
- Automatically set view script, helper, and filter paths based on the current module, and automatically associate the current module name as a class prefix for helper and filter classes.
- Create a globally available view object for all dispatched controllers and actions.
- Allow the developer to set default view rendering options for all controllers.
- Add the ability to automatically render a view script with no intervention.
- Allow the developer to create her own specifications for the view base path and for view script paths.

  Note: If you perform a _forward(), redirect(), or render() manually, autorendering will not occur, as by performing any of these actions you are telling the *ViewRenderer* that you are determining your own output.

  Note: The *ViewRenderer* is enabled by default. You may disable it via the front controller *noViewRenderer* param (**$front->setParam('noViewRenderer', true);**) or removing the helper from the helper broker stack (Zend_Controller_Action_HelperBroker::removeHelper('viewRenderer')).
  If you wish to modify settings of the *ViewRenderer* prior to dispatching the front controller, you may do so in one of two ways:

  - Instantiate and register your own *ViewRenderer* object and pass it to the helper broker:

    ```
    $viewRenderer = new
    Zend_Controller_Action_Helper_ViewRenderer();
    $viewRenderer->setView($view)
                 ->setViewSuffix('php');
    Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
    ```

  - Initialize and/or retrieve a *ViewRenderer* object on demand via the helper broker:

    ```
    $viewRenderer =

    Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
    $viewRenderer->setView($view)
                 ->setViewSuffix('php');
    ```

API

At its most basic usage, you simply instantiate the *ViewRenderer* and pass it to the action helper broker. The easiest way to instantiate it and register in one go is to use the helper broker's getStaticHelper() method:

```
Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
```

The first time an action controller is instantiated, it will trigger the *ViewRenderer* to instantiate a view object. Each time

a controller is instantiated, the *ViewRenderer*'s init() method is called, which will cause it to set the view property of the action controller, and call addScriptPath() with a path relative to the current module; this will be called with a class prefix named after the current module, effectively namespacing all helper and filter classes you define for the module.

Each time postDispatch() is called, it will call render() for the current action.

As an example, consider the following class:

```
1.  // A controller class, foo module:
2.  class Foo_BarController extends Zend_Controller_Action
3.  {
4.      // Render bar/index.phtml by default; no action required
5.      public function indexAction()
6.      {
7.      }
8.
9.      // Render bar/populate.phtml with variable 'foo' set to 'bar'.
10.     // Since view object defined at preDispatch(), it's already available.
11.     public function populateAction()
12.     {
13.         $this->view->foo = 'bar';
14.     }
15. }
16.
17. ...
18.
19. // in one of your view scripts:
20. $this->foo(); // call Foo_View_Helper_Foo::foo()
```

The *ViewRenderer* also defines a number of accessors to allow setting and retrieving view options:

- setView($view) allows you to set the view object for the *ViewRenderer*. It gets set as the public class property *$view*.
- setNeverRender($flag = true) can be used to disable or enable autorendering globally, i.e., for all controllers. If set to **TRUE**, postDispatch() will not automatically call render() in the current controller. getNeverRender() retrieves the current value.
- setNoRender($flag = true) can be used to disable or enable autorendering. If set to **TRUE**, postDispatch() will not automatically call render() in the current controller. This setting is reset each time preDispatch() is called (i.e., you need to set this flag for each controller for which you don't want autorenderering to occur). getNoRender() retrieves the current value.
- setNoController($flag = true) can be used to tell render() not to look for the action script in a subdirectory named after the controller (which is the default behaviour). getNoController() retrieves the current value.
- setNeverController($flag = true) is analogous to setNoController(), but works on a global level -- i.e., it will not be reset for each dispatched action. getNeverController() retrieves the current value.
- setScriptAction($name) can be used to specify the action script to render. *$name* should be the name of the script minus the file suffix (and without the controller subdirectory, unless *noController* has been turned on). If not specified, it looks for a view script named after the action in the request object. getScriptAction() retrieves the current value.
- setResponseSegment($name) can be used to specify which response object named segment to render into. If not specified, it renders into the default segment. getResponseSegment() retrieves the current value.
- initView($path, $prefix, $options) may be called to specify the base view path, class prefix for helper and filter scripts, and *ViewRenderer* options. You may pass any of the following flags: *neverRender*, *noRender*,

noController, *scriptAction*, and *responseSegment*.

- setRender($action = null, $name = null, $noController = false) allows you to set any of *scriptAction*, *responseSegment*, and *noController* in one pass. direct() is an alias to this method, allowing you to call this method easily from your controller:

```
// Render 'foo' instead of current action script
$this->_helper->viewRenderer('foo');

// render form.phtml to the 'html' response segment, without using a
// controller view script subdirectory:
$this->_helper->viewRenderer('form', 'html', true);
```

  Note: setRender() and direct() don't actually render the view script, but instead set hints that postDispatch() and render() will use to render the view.

The constructor allows you to optionally pass the view object and *ViewRenderer* options; it accepts the same flags as initView():

```
$view    = new Zend_View(array('encoding' => 'UTF-8'));
$options = array('noController' => true, 'neverRender' => true);
$viewRenderer =
    new Zend_Controller_Action_Helper_ViewRenderer($view, $options);
```

There are several additional methods for customizing path specifications used for determining the view base path to add to the view object, and the view script path to use when autodetermining the view script to render. These methods each take one or more of the following placeholders:

- *:moduleDir* refers to the current module's base directory (by convention, the parent directory of the module's controller directory).
- *:module* refers to the current module name.
- *:controller* refers to the current controller name.
- *:action* refers to the current action name.
- *:suffix* refers to the view script suffix (which may be set via setViewSuffix()).

The methods for controlling path specifications are:

- setViewBasePathSpec($spec) allows you to change the path specification used to determine the base path to add to the view object. The default specification is *:moduleDir/views*. You may retrieve the current specification at any time using getViewBasePathSpec().
- setViewScriptPathSpec($spec) allows you to change the path specification used to determine the path to an individual view script (minus the base view script path). The default specification is *:controller/:action.:suffix*. You may retrieve the current specification at any time using getViewScriptPathSpec().
- setViewScriptPathNoControllerSpec($spec) allows you to change the path specification used to determine the path to an individual view script when *noController* is in effect (minus the base view script path). The default specification is *:action.:suffix*. You may retrieve the current specification at any time using getViewScriptPathNoControllerSpec().

For fine-grained control over path specifications, you may use Zend_Filter_Inflector. Under the hood, the *ViewRenderer* uses an inflector to perform path mappings already. To interact with the inflector -- either to set your own for use, or to modify the default inflector, the following methods may be used:

- getInflector() will retrieve the inflector. If none exists yet in the *ViewRenderer*, it creates one using the default rules.

By default, it uses static rule references for the suffix and module directory, as well as a static target; this allows various *ViewRenderer* properties the ability to dynamically modify the inflector.

- setInflector($inflector, $reference) allows you to set a custom inflector for use with the *ViewRenderer*. If *$reference* is `TRUE`, it will set the suffix and module directory as static references to *ViewRenderer* properties, as well as the target.

  Note: **Default Lookup Conventions**
  The *ViewRenderer* does some path normalization to make view script lookups easier. The default rules are as follows:

  - *:module*: MixedCase and camelCasedWords are separated by dashes, and the entire string cast to lowercase. E.g.: "FooBarBaz" becomes "foo-bar-baz".

    Internally, the inflector uses the filters Zend_Filter_Word_CamelCaseToDash and Zend_Filter_StringToLower.

  - *:controller*: MixedCase and camelCasedWords are separated by dashes; underscores are converted to directory separators, and the entire string cast to lower case. Examples: "FooBar" becomes "foo-bar"; "FooBar_Admin" becomes "*foo-bar/admin*".

    Internally, the inflector uses the filters Zend_Filter_Word_CamelCaseToDash, Zend_Filter_Word_UnderscoreToSeparator, and Zend_Filter_StringToLower.

  - *:action*: MixedCase and camelCasedWords are separated by dashes; non-alphanumeric characters are translated to dashes, and the entire string cast to lower case. Examples: "fooBar" becomes "foo-bar"; "foo-barBaz" becomes "foo-bar-baz".

    Internally, the inflector uses the filters Zend_Filter_Word_CamelCaseToDash, Zend_Filter_PregReplace, and Zend_Filter_StringToLower.

The final items in the *ViewRenderer* API are the methods for actually determining view script paths and rendering views. These include:

- renderScript($script, $name) allows you to render a script with a path you specify, optionally to a named path segment. When using this method, the *ViewRenderer* does no autodetermination of the script name, but instead directly passes the *$script* argument directly to the view object's render() method.

  Note: Once the view has been rendered to the response object, it sets the *noRender* to prevent accidentally rendering the same view script multiple times.

  Note: By default, Zend_Controller_Action::renderScript() proxies to the *ViewRenderer*'s renderScript() method.

- getViewScript($action, $vars) creates the path to a view script based on the action passed and/or any variables passed in *$vars*. Keys for this array may include any of the path specification keys ('moduleDir', 'module', 'controller', 'action', and 'suffix'). Any variables passed will be used; otherwise, values based on the current request will be utlized.

  getViewScript() will use either the *viewScriptPathSpec* or *viewScriptPathNoControllerSpec* based on the setting of the *noController* flag.

  Word delimiters occurring in module, controller, or action names will be replaced with dashes ('-'). Thus, if you have the controller name '**foo.bar**' and the action '**baz:bat**', using the default path specification will result in a view script path of '*foo-bar/baz-bat.phtml*'.

  Note: By default, Zend_Controller_Action::getViewScript() proxies to the *ViewRenderer*'s getViewScript() method.

- render($action, $name, $noController) checks first to see if either *$name* or *$noController* have been passed, and

if so, sets the appropriate flags (responseSegment and noController, respectively) in the ViewRenderer. It then passes the *$action* argument, if any, on to getViewScript(). Finally, it passes the calculated view script path to renderScript().

> **Note**: Be aware of the side-effects of using render(): the values you pass for the response segment name and for the noController flag will persist in the object. Additionally, noRender will be set after rendering is completed.
>
> **Note**: By default, Zend_Controller_Action::render() proxies to the *ViewRenderer*'s render() method.

- renderBySpec($action, $vars, $name) allows you to pass path specification variables in order to determine the view script path to create. It passes *$action* and *$vars* to getScriptPath(), and then passes the resulting script path and *$name* on to renderScript().

Basic Usage Examples

Example #9 Basic Usage

At its most basic, you simply initialize and register the *ViewRenderer* helper with the helper broker in your bootstrap, and then set variables in your action methods.

```
// In your bootstrap:
Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');

...

// 'foo' module, 'bar' controller:
class Foo_BarController extends Zend_Controller_Action
{
    // Render bar/index.phtml by default; no action required
    public function indexAction()
    {
    }

    // Render bar/populate.phtml with variable 'foo' set to 'bar'.
    // Since view object defined at preDispatch(), it's already available.
    public function populateAction()
    {
        $this->view->foo = 'bar';
    }

    // Renders nothing as it forwards to another action; the new action
    // will perform any rendering
    public function bazAction()
    {
        $this->_forward('index');
    }

    // Renders nothing as it redirects to another location
    public function batAction()
    {
        $this->_redirect('/index');
    }
}
```

> **Note**: **Naming Conventions: Word Delimiters in Controller and Action Names**
> If your controller or action name is composed of several words, the dispatcher requires that these are separated on the URL by specific path and word delimiter characters. The *ViewRenderer* replaces any path

delimiter found in the controller name with an actual path delimiter ('/'), and any word delimiter found with a dash ('-') when creating paths. Thus, a call to the action */foo.bar/baz.bat* would dispatch to FooBarController::bazBatAction() in *FooBarController.php*, which would render *foo-bar/baz-bat.phtml*; a call to the action */bar_baz/baz-bat* would dispatch to Bar_BazController::bazBatAction() in *Bar/BazController.php* (note the path separation) and render *bar/baz/baz-bat.phtml*.
Note that the in the second example, the module is still the default module, but that, because of the existence of a path separator, the controller receives the name Bar_BazController, in *Bar/BazController.php*. The ViewRenderer mimics the controller directory hierarchy.

Example #10 Disabling Autorender

For some actions or controllers, you may want to turn off the autorendering -- for instance, if you're wanting to emit a different type of output (XML, JSON, etc), or if you simply want to emit nothing. You have two options: turn off all cases of autorendering (setNeverRender()), or simply turn it off for the current action (setNoRender()).

```
// Baz controller class, bar module:
class Bar_BazController extends Zend_Controller_Action
{
    public function fooAction()
    {
        // Don't auto render this action
        $this->_helper->viewRenderer->setNoRender();
    }
}

// Bat controller class, bar module:
class Bar_BatController extends Zend_Controller_Action
{
    public function preDispatch()
    {
        // Never auto render this controller's actions
        $this->_helper->viewRenderer->setNoRender();
    }
}
```

Note: In most cases, it makes no sense to turn off autorendering globally (ala setNeverRender()), as the only thing you then gain from *ViewRenderer* is the autosetup of the view object.

Example #11 Choosing a Different View Script

Some situations require that you render a different script than one named after the action. For instance, if you have a controller that has both add and edit actions, they may both display the same 'form' view, albeit with different values set. You can easily change the script name used with either setScriptAction(), setRender(), or calling the helper as a method, which will invoke setRender().

```
// Bar controller class, foo module:
class Foo_BarController extends Zend_Controller_Action
{
    public function addAction()
    {
        // Render 'bar/form.phtml' instead of 'bar/add.phtml'
        $this->_helper->viewRenderer('form');
    }

    public function editAction()
    {
        // Render 'bar/form.phtml' instead of 'bar/edit.phtml'
        $this->_helper->viewRenderer->setScriptAction('form');
```

```
    }

    public function processAction()
    {
        // do some validation...
        if (!$valid) {
            // Render 'bar/form.phtml' instead of 'bar/process.phtml'
            $this->_helper->viewRenderer->setRender('form');
            return;
        }

        // otherwise continue processing...
    }

}
```

Example #12 Modifying the Registered View

What if you need to modify the view object -- for instance, change the helper paths, or the encoding? You can do so either by modifying the view object set in your controller, or by grabbing the view object out of the *ViewRenderer*; both are references to the same object.

```
// Bar controller class, foo module:
class Foo_BarController extends Zend_Controller_Action
{
    public function preDispatch()
    {
        // change view encoding
        $this->view->setEncoding('UTF-8');
    }

    public function bazAction()
    {
        // Get view object and set escape callback to 'htmlspecialchars'
        $view = $this->_helper->viewRenderer->view;
        $view->setEscape('htmlspecialchars');
    }
}
```

Advanced Usage Examples

Example #13 Changing the Path Specifications

In some circumstances, you may decide that the default path specifications do not fit your site's needs. For instance, you may want to have a single template tree to which you may then give access to your designers (this is very typical when using » Smarty, for instance). In such a case, you may want to hardcode the view base path specification, and create an alternate specification for the action view script paths themselves.

For purposes of this example, let's assume that the base path to views should be '*/opt/vendor/templates*', and that you wish for view scripts to be referenced by '*:moduleDir/:controller/:action.:suffix*'; if the *noController* flag has been set, you want to render out of the top level instead of in a subdirectory ('*:action.:suffix*'). Finally, you want to use 'tpl' as the view script filename suffix.

```
/**
* In your bootstrap:
*/

// Different view implementation
```

```
$view = new ZF_Smarty();

$viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer($view);
$viewRenderer->setViewBasePathSpec('/opt/vendor/templates')
             ->setViewScriptPathSpec(':module/:controller/:action.:suffix')
             ->setViewScriptPathNoControllerSpec(':action.:suffix')
             ->setViewSuffix('tpl');
Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
```

Example #14 Rendering Multiple View Scripts from a Single Action

At times, you may need to render multiple view scripts from a single action. This is very straightforward -- simply make multiple calls to render():

```
class SearchController extends Zend_Controller_Action
{
    public function resultsAction()
    {
        // Assume $this->model is the current model
        $this->view->results =
            $this->model->find($this->_getParam('query', ''));

        // render() by default proxies to the ViewRenderer
        // Render first the search form and then the results
        $this->render('form');
        $this->render('results');
    }

    public function formAction()
    {
        // do nothing; ViewRenderer autorenders the view script
    }
}
```

Writing Your Own Helpers

Action helpers extend Zend_Controller_Action_Helper_Abstract, an abstract class that provides the basic interface and functionality required by the helper broker. These include the following methods:

- setActionController() is used to set the current action controller.
- init(), triggered by the helper broker at instantiation, can be used to trigger initialization in the helper; this can be useful for resetting state when multiple controllers use the same helper in chained actions.
- preDispatch(), is triggered prior to a dispatched action.
- postDispatch() is triggered when a dispatched action is done -- even if a preDispatch() plugin has skipped the action. Mainly useful for cleanup.
- getRequest() retrieves the current request object.
- getResponse() retrieves the current response object.
- getName() retrieves the helper name. It retrieves the portion of the class name following the last underscore character, or the full class name otherwise. As an example, if the class is named Zend_Controller_Action_Helper_Redirector, it will return *Redirector*; a class named *FooMessage* will simply return itself.

You may optionally include a direct() method in your helper class. If defined, it allows you to treat the helper as a method of the helper broker, in order to allow easy, one-off usage of the helper. As an example, the redirector defines direct() as an alias of goto(), allowing use of the helper like this:

```
1. // Redirect to /blog/view/item/id/42
2. $this->_helper->redirector('item', 'view', 'blog', array('id' => 42));
```

Internally, the helper broker's __call() method looks for a helper named *redirector*, then checks to see if that helper has a defined direct() method, and calls it with the arguments provided.

Once you have created your own helper class, you may provide access to it as described in the sections above.

- Action Controllers
- The Response Object

bootstrap"""

I have not initialized and registered the ViewRenderer helper anywhere. It is done automatically by zend framework somewhere. I have been following Keith Pope's book and have ony defined "resources.view[] = " in the application.ini file.My application runs well. Any comments from anybody please... "resources.view[] = " in the application.ini file.My application runs well. Any comments from anybody please...

Reply • Share ›

devvyn ➚ Dkps3012 • 2 years ago

I believe you're correct. ZF does almost everything automatically, but there are times when the official documentation is ambiguous about the behaviour your should expect.

Reply • Share ›

Dominik Sipowicz ➚ Dkps3012 • 2 years ago

Yes, you did it exacly by "resources.view[] = "

Reply • Share ›

Get Started

- Downloads
- Get the Skeleton App
- Find modules for your application

Participate

- Overview
- Contributors Guide
- Blogs

Contact Us

- form
- mailing lists
- IRC