Zend\View Quick Start[¶](http://www.zendframework.com/manual/2.2/en/modules/zend.view.quick-start.html#zend-view-quick-start)

Overview

Zend\View provides the “View” layer of Zend Framework 2’s MVC system. It is a multi-tiered system allowing a variety of mechanisms for extension, substitution, and more.

The components of the view layer are as follows:

Zend\View 提供了“View”层 Zend Framework 2’的 MVC 系统，他是一个多层次的系统，允许各种各样对机制的扩展，代替等等。

View层的各种组件如下：

1. **变量容器（Variables Containers）**保存你想要展现在页面上的变量和回调函数（callbacks）。通常，变量容器（Variables Containers）也会提供 机制对上下文的变量的转义等等。

**Variables Containers** hold variables and callbacks that you wish to represent in the view. Often-times, a Variables Container will also provide mechanisms for context-specific escaping of variables and more.

1. **视图模型（View Models）**保存变量容器，指定模板来使用（如果有的话），选择的 提供 渲染 设置项。可以嵌套视图模型来代表复杂的结构。

* **View Models** hold Variables Containers, specify the template to use (if any), and optionally provide rendering options (more on that below). View Models may be nested in order to represent complex structures.

1. **渲染器（Renderers）**获取视图模型并且提供他们的表示的返回。ZF2默认提供3种渲染器：PhpRenderer负责利用PHP模板生成标记，JsonRenderer和FeedRenderer生成RSS和Atom.

* **Renderers** take View Models and provide a representation of them to return. Zend Framework 2 ships with three renderers by default: a PhpRenderer which utilizes PHP templates in order to generate markup, a JsonRenderer, and a FeedRenderer for generating RSS and Atom feeds.

**解析器（Resolvers）**

* **Resolvers** utilizes Resolver Strategies to resolve a template name to a resource a Renderer may consume. As an example, a Resolver may take the name “blog/entry” and resolve it to a PHP view script.
* The **View** consists of strategies that map the current Request to a Renderer, and strategies for injecting the result of rendering to the Response.
* **Rendering Strategies** listen to the Zend\View\ViewEvent::EVENT\_RENDERER event of the View and decide which Renderer should be selected based on the Request or other criteria.
* **Response Strategies** are used to inject the Response object with the results of rendering. That may also include taking actions such as setting Content-Type headers.

Additionally, Zend Framework 2 provides integration with the MVC via a number of event listeners in the Zend\Mvc\View namespace.

Usage[¶](http://www.zendframework.com/manual/2.2/en/modules/zend.view.quick-start.html#usage)

This section of the manual is designed to show you typical usage patterns of the view layer when using it within the Zend Framework 2 MVC. The assumptions are that you are using [*Dependency Injection*](http://www.zendframework.com/manual/2.2/en/index.html#zend-di) and the default MVC view strategies.

Configuration[¶](http://www.zendframework.com/manual/2.2/en/modules/zend.view.quick-start.html#configuration)

The default configuration will typically work out-of-the-box. However, you will still need to select Resolver Strategies and configure them, as well as potentially indicate alternate template names for things like the site layout, 404 (not found) pages, and error pages. The code snippets below can be added to your configuration to accomplish this. We recommend adding it to a site-specific module, such as the “Application” module from the framework’s [ZendSkeletonApplication](https://github.com/zendframework/ZendSkeletonApplication), or to one of your autoloaded configurations within theconfig/autoload/ directory.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 | **return** **array**(  'view\_manager' => **array**(  *// The TemplateMapResolver allows you to directly map template names*  *// to specific templates. The following map would provide locations*  *// for a home page template ("application/index/index"), as well as for*  *// the layout ("layout/layout"), error pages ("error/index"), and*  *// 404 page ("error/404"), resolving them to view scripts.*  'template\_map' => **array**(  'application/index/index' => \_\_DIR\_\_ . '/../view/application/index/index.phtml',  'site/layout' => \_\_DIR\_\_ . '/../view/layout/layout.phtml',  'error/index' => \_\_DIR\_\_ . '/../view/error/index.phtml',  'error/404' => \_\_DIR\_\_ . '/../view/error/404.phtml',  ),  *// The TemplatePathStack takes an array of directories. Directories*  *// are then searched in LIFO order (it's a stack) for the requested*  *// view script. This is a nice solution for rapid application*  *// development, but potentially introduces performance expense in*  *// production due to the number of static calls necessary.*  *//*  *// The following adds an entry pointing to the view directory*  *// of the current module. Make sure your keys differ between modules*  *// to ensure that they are not overwritten -- or simply omit the key!*  'template\_path\_stack' => **array**(  'application' => \_\_DIR\_\_ . '/../view',  ),  *// This will be used as the default suffix for template scripts resolving, it defaults to 'phtml'.*  'default\_template\_suffix' => 'php',  *// Set the template name for the site's layout.*  *//*  *// By default, the MVC's default Rendering Strategy uses the*  *// template name "layout/layout" for the site's layout.*  *// Here, we tell it to use the "site/layout" template,*  *// which we mapped via the TemplateMapResolver above.*  'layout' => 'site/layout',  *// By default, the MVC registers an "exception strategy", which is*  *// triggered when a requested action raises an exception; it creates*  *// a custom view model that wraps the exception, and selects a*  *// template. We'll set it to "error/index".*  *//*  *// Additionally, we'll tell it that we want to display an exception*  *// stack trace; you'll likely want to disable this by default.*  'display\_exceptions' => **true**,  'exception\_template' => 'error/index',  *// Another strategy the MVC registers by default is a "route not*  *// found" strategy. Basically, this gets triggered if (a) no route*  *// matches the current request, (b) the controller specified in the*  *// route match cannot be found in the service locator, (c) the controller*  *// specified in the route match does not implement the DispatchableInterface*  *// interface, or (d) if a response from a controller sets the*  *// response status to 404.*  *//*  *// The default template used in such situations is "error", just*  *// like the exception strategy. Here, we tell it to use the "error/404"*  *// template (which we mapped via the TemplateMapResolver, above).*  *//*  *// You can opt in to inject the reason for a 404 situation; see the*  *// various `Application\:\:ERROR\_\*`\_ constants for a list of values.*  *// Additionally, a number of 404 situations derive from exceptions*  *// raised during routing or dispatching. You can opt-in to display*  *// these.*  'display\_not\_found\_reason' => **true**,  'not\_found\_template' => 'error/404',  ),  ); |

Controllers and View Models[¶](http://www.zendframework.com/manual/2.2/en/modules/zend.view.quick-start.html#controllers-and-view-models)

Zend\View\View consumes ViewModels, passing them to the selected renderer. Where do you create these, though?

The most explicit way is to create them in your controllers and return them.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **namespace** Foo\Controller;  **use** Zend\Mvc\Controller\AbstractActionController;  **use** Zend\View\Model\ViewModel;  **class** **BazBatController** **extends** AbstractActionController  {  **public** **function** doSomethingCrazyAction()  {  $view = **new** ViewModel(**array**(  'message' => 'Hello world',  ));  $view->setTemplate('foo/baz-bat/do-something-crazy');  **return** $view;  }  } |

This sets a “message” variable in the View Model, and sets the template name “foo/baz-bat/do-something-crazy”. The View Model is then returned.

In most cases, you’ll likely have a template name based on the module namespace, controller, and action. Considering that, and if you’re simply passing some variables, could this be made simpler? Definitely.

The MVC registers a couple of listeners for controllers to automate this. The first will look to see if you returned an associative array from your controller; if so, it will create a View Model and make this associative array the Variables Container; this View Model then replaces the[*MvcEvent*](http://www.zendframework.com/manual/2.2/en/modules/zend.mvc.mvc-event.html#zend-mvc-mvc-event)‘s result. It will also look to see if you returned nothing or null; if so, it will create a View Model without any variables attached; this View Model also replaces the MvcEvent‘s result.

The second listener checks to see if the MvcEvent result is a View Model, and, if so, if it has a template associated with it. If not, it will inspect the controller matched during routing to determine the module namespace and the controller class name, and, if available, it’s “action” parameter in order to create a template name. This will be “module/controller/action”, all normalized to lowercase, dash-separated words.

As an example, the controller Foo\Controller\BazBatController with action “doSomethingCrazyAction”, would be mapped to the template foo/baz-bat/do-something-crazy. As you can see, the words “Controller” and “Action” are omitted.

In practice, that means our previous example could be re-written as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **namespace** Foo\Controller;  **use** Zend\Mvc\Controller\AbstractActionController;  **class** **BazBatController** **extends** AbstractActionController  {  **public** **function** doSomethingCrazyAction()  {  **return** **array**(  'message' => 'Hello world',  );  }  } |

The above method will likely work for the majority of use cases. When you need to specify a different template, explicitly create and return a View Model and specify the template manually, as in the first example.

Nesting View Models[¶](http://www.zendframework.com/manual/2.2/en/modules/zend.view.quick-start.html#nesting-view-models)

The other use case you may have for setting explicit View Models is if you wish to **nest** them. In other words, you might want to render templates to be included within the main View you return.

As an example, you may want the View from an action to be one primary section that includes both an “article” and a couple of sidebars; one of the sidebars may include content from multiple Views as well:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | **namespace** Content\Controller;  **use** Zend\Mvc\Controller\AbstractActionController;  **use** Zend\View\Model\ViewModel;  **class** **ArticleController** **extends** AbstractActionController  {  **public** **function** viewAction()  {  *// get the article from the persistence layer, etc...*  $view = **new** ViewModel();  *// this is not needed since it matches "module/controller/action"*  $view->setTemplate('content/article/view');  $articleView = **new** ViewModel(**array**('article' => $article));  $articleView->setTemplate('content/article');  $primarySidebarView = **new** ViewModel();  $primarySidebarView->setTemplate('content/main-sidebar');  $secondarySidebarView = **new** ViewModel();  $secondarySidebarView->setTemplate('content/secondary-sidebar');  $sidebarBlockView = **new** ViewModel();  $sidebarBlockView->setTemplate('content/block');  $secondarySidebarView->addChild($sidebarBlockView, 'block');  $view->addChild($articleView, 'article')  ->addChild($primarySidebarView, 'sidebar\_primary')  ->addChild($secondarySidebarView, 'sidebar\_secondary');  **return** $view;  }  } |

The above will create and return a View Model specifying the template “content/article/view”. When the View is rendered, it will render three child Views, the $articleView, $primarySidebarView, and $secondarySidebarView; these will be captured to the $view‘s “article”, “sidebar\_primary”, and “sidebar\_secondary” variables, respectively, so that when it renders, you may include that content. Additionally, the $secondarySidebarView will include an additional View Model, $sidebarBlockView, which will be captured to its “block” view variable.

To better visualize this, let’s look at what the final content might look like, with comments detailing where each nested view model is injected.

Here are the templates, rendered based on a 12-column grid:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <?php *// "content/article/view" template ?>*  <!-- **This** is from the $view View Model, **and** the "content/article/view" template -->  <div class="row content">  <?php **echo** $this->article ?>  <?php **echo** $this->sidebar\_primary ?>  <?php **echo** $this->sidebar\_secondary ?>  </div> |
| 1  2  3  4  5  6 | <?php *// "content/article" template ?>*  <!-- **This** is from the $articleView View Model, **and** the "content/article"  template -->  <article class="span8">  <?php **echo** $this->escapeHtml('article') ?>  </article> |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <?php *// "content/main-sidebar" template ?>*  <!-- **This** is from the $primarySidebarView View Model, **and** the  "content/main-sidebar" template -->  <div class="span2 sidebar">  sidebar content...  </div> |
| 1  2  3  4  5  6 | <?php *// "content/secondary-sidebar template ?>*  <!-- **This** is from the $secondarySidebarView View Model, **and** the  "content/secondary-sidebar" template -->  <div class="span2 sidebar pull-right">  <?php **echo** $this->block ?>  </div> |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <?php *// "content/block template ?>*  <!-- **This** is from the $sidebarBlockView View Model, **and** the  "content/block" template -->  <div class="block">  block content...  </div> |

And here is the aggregate, generated content:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | *<!-- This is from the $view View Model, and the "content/article/view" template -->*  **<div** class="row content"**>**  *<!-- This is from the $articleView View Model, and the "content/article"*  *template -->*  **<article** class="span8"**>**  Lorem ipsum ....  **</article>**  *<!-- This is from the $primarySidebarView View Model, and the*  *"content/main-sidebar" template -->*  **<div** class="span2 sidebar"**>**  sidebar content...  **</div>**  *<!-- This is from the $secondarySidebarView View Model, and the*  *"content/secondary-sidebar" template -->*  **<div** class="span2 sidebar pull-right"**>**  *<!-- This is from the $sidebarBlockView View Model, and the*  *"content/block" template -->*  **<div** class="block"**>**  block content...  **</div>**  **</div>**  **</div>** |

As you can see, you can achieve very complex markup using nested Views, while simultaneously keeping the details of rendering isolated from the Request/Response lifecycle of the controller.

Dealing with Layouts[¶](http://www.zendframework.com/manual/2.2/en/modules/zend.view.quick-start.html#dealing-with-layouts)

Most sites enforce a cohesive look-and-feel which we typically call the site’s “layout”. It includes the default stylesheets and JavaScript necessary, if any, as well as the basic markup structure into which all site content will be injected.

Within Zend Framework 2, layouts are handled via nesting of View Models (see the [*previous example*](http://www.zendframework.com/manual/2.2/en/modules/zend.view.quick-start.html#zend-view-quick-start-usage-nesting) for examples of View Model nesting). The Zend\Mvc\View\Http\ViewManager composes a View Model which acts as the “root” for nested View Models. As such, it should contain the skeleton (or layout) template for the site. All other content is then rendered and captured to view variables of this root View Model.

The ViewManager sets the layout template as “layout/layout” by default. To change this, you can add some configuration to the “view\_manager” area of your [*configuration*](http://www.zendframework.com/manual/2.2/en/modules/zend.view.quick-start.html#zend-view-quick-start-usage-config).

A listener on the controllers, Zend\Mvc\View\Http\InjectViewModelListener, will take a View Model returned from a controller and inject it as a child of the root (layout) View Model. By default, View Models will capture to the “content” variable of the root View Model. This means you can do the following in your layout view script:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <html>  <head>  <title><?php **echo** $this->headTitle() ?></title>  </head>  <body>  <?php **echo** $this->content; ?>  </body>  </html> |

If you want to specify a different View variable for which to capture, explicitly create a view model in your controller, and set its “capture to” value:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **namespace** Foo\Controller;  **use** Zend\Mvc\Controller\AbstractActionController;  **use** Zend\View\Model\ViewModel;  **class** **BazBatController** **extends** AbstractActionController  {  **public** **function** doSomethingCrazyAction()  {  $view = **new** ViewModel(**array**(  'message' => 'Hello world',  ));  *// Capture to the layout view's "article" variable*  $view->setCaptureTo('article');  **return** $view;  }  } |

There will be times you don’t want to render a layout. For example, you might be answering an API call which expects JSON or an XML payload, or you might be answering an XHR request that expects a partial HTML payload. The simplest way to do this is to explicitly create and return a view model from your controller, and mark it as “terminal”, which will hint to the MVC listener that normally injects the returned View Model into the layout View Model, to instead replace the layout view model.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **namespace** Foo\Controller;  **use** Zend\Mvc\Controller\AbstractActionController;  **use** Zend\View\Model\ViewModel;  **class** **BazBatController** **extends** AbstractActionController  {  **public** **function** doSomethingCrazyAction()  {  $view = **new** ViewModel(**array**(  'message' => 'Hello world',  ));  *// Disable layouts; `MvcEvent` will use this View Model instead*  $view->setTerminal(**true**);  **return** $view;  }  } |

[*When discussing nesting View Models*](http://www.zendframework.com/manual/2.2/en/modules/zend.view.quick-start.html#zend-view-quick-start-usage-nesting), we detailed a nested View Model which contained an article and sidebars. Sometimes, you may want to provide additional View Models to the layout, instead of nesting in the returned layout. This may be done by using the “layout” controller plugin, which returns the root View Model. You can then call the same addChild() method on it as we did in that previous example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **namespace** Content\Controller;  **use** Zend\Mvc\Controller\AbstractActionController;  **use** Zend\View\Model\ViewModel;  **class** **ArticleController** **extends** AbstractActionController  {  **public** **function** viewAction()  {  *// get the article from the persistence layer, etc...*  *// Get the "layout" view model and inject a sidebar*  $layout = $this->layout();  $sidebarView = **new** ViewModel();  $sidebarView->setTemplate('content/sidebar');  $layout->addChild($sidebarView, 'sidebar');  *// Create and return a view model for the retrieved article*  $view = **new** ViewModel(**array**('article' => $article));  $view->setTemplate('content/article');  **return** $view;  }  } |

You could also use this technique to select a different layout, by simply calling the setTemplate() method of the layout View Model:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | *//In a controller*  **namespace** Content\Controller;  **use** Zend\Mvc\Controller\AbstractActionController;  **use** Zend\View\Model\ViewModel;  **class** **ArticleController** **extends** AbstractActionController  {  **public** **function** viewAction()  {  *// get the article from the persistence layer, etc...*  *// Get the "layout" view model and set an alternate template*  $layout = $this->layout();  $layout->setTemplate('article/layout');  *// Create and return a view model for the retrieved article*  $view = **new** ViewModel(**array**('article' => $article));  $view->setTemplate('content/article');  **return** $view;  }  } |

Sometimes, you may want to access the layout from within your actual view scripts when using the PhpRenderer. Reasons might include wanting to change the layout template or wanting to either access or inject layout view variables. Similar to the “layout” controller plugin, you can use the “layout” View Helper. If you provide a string argument to it, you will change the template; if you provide no arguments, the root layout View Model is returned.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | *//In a view script*  *// Change the layout:*  $this->layout('alternate/layout'); *// OR*  $this->layout()->setTemplate('alternate/layout');  *// Access a layout variable.*  *// Since access to the base view model is relatively easy, it becomes a*  *// reasonable place to store things such as API keys, which other view scripts*  *// may need.*  $layout = $this->layout();  $disqusApiKey = **false**;  **if** (isset($layout->disqusApiKey)) {  $disqusApiKey = $layout->disqusApiKey;  }  *// Set a layout variable*  $this->layout()->footer = $this->render('article/footer'); |

Commonly, you may want to alter the layout based on the current **module**. This requires (a) detecting if the controller matched in routing belongs to this module, and then (b) changing the template of the View Model.

The place to do these actions is in a listener. It should listen either to the “route” event at low (negative) priority, or on the “dispatch” event, at any priority. Typically, you will register this during the bootstrap event.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | **namespace** Content;  **class** **Module**  {  */\*\**  *\* @param \Zend\Mvc\MvcEvent $e The MvcEvent instance*  *\* @return void*  *\*/*  **public** **function** onBootstrap($e)  {  *// Register a dispatch event*  $app = $e->getParam('application');  $app->getEventManager()->attach('dispatch', **array**($this, 'setLayout'));  }  */\*\**  *\* @param \Zend\Mvc\MvcEvent $e The MvcEvent instance*  *\* @return void*  *\*/*  **public** **function** setLayout($e)  {  $matches = $e->getRouteMatch();  $controller = $matches->getParam('controller');  **if** (**false** === strpos($controller, \_\_NAMESPACE\_\_)) {  *// not a controller from this module*  **return**;  }  *// Set the layout template*  $viewModel = $e->getViewModel();  $viewModel->setTemplate('content/layout');  }  } |

Creating and Registering Alternate Rendering and Response Strategies[¶](http://www.zendframework.com/manual/2.2/en/modules/zend.view.quick-start.html#creating-and-registering-alternate-rendering-and-response-strategies)

Zend\View\View does very little. Its workflow is essentially to martial a ViewEvent, and then trigger two events, “renderer” and “response”. You can attach “strategies” to these events, using the methods addRenderingStrategy() and addResponseStrategy(), respectively. A Rendering Strategy investigates the Request object (or any other criteria) in order to select a Renderer (or fail to select one). A Response Strategy determines how to populate the Response based on the result of rendering.

Zend Framework 2 ships with three Rendering and Response Strategies that you can use within your application.

* Zend\View\Strategy\PhpRendererStrategy. This strategy is a “catch-all” in that it will always return theZend\View\Renderer\PhpRenderer and populate the Response body with the results of rendering.
* Zend\View\Strategy\JsonStrategy. This strategy inspects the Accept HTTP header, if present, and determines if the client has indicated it accepts an “application/json” response. If so, it will return the Zend\View\Renderer\JsonRenderer, and populate the Response body with the JSON value returned, as well as set a Content-Type header with a value of “application/json”.
* Zend\View\Strategy\FeedStrategy. This strategy inspects the Accept HTTP header, if present, and determines if the client has indicated it accepts either an “application/rss+xml” or “application/atom+xml” response. If so, it will return theZend\View\Renderer\FeedRenderer, setting the feed type to either “rss” or “atom”, based on what was matched. Its Response strategy will populate the Response body with the generated feed, as well as set a Content-Type header with the appropriate value based on feed type.

By default, only the PhpRendererStrategy is registered, meaning you will need to register the other Strategies yourself if you want to use them. Additionally, it means that you will likely want to register these at higher priority to ensure they match before thePhpRendererStrategy. As an example, let’s register the JsonStrategy:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | **namespace** Application;  **class** **Module**  {  */\*\**  *\* @param \Zend\Mvc\MvcEvent $e The MvcEvent instance*  *\* @return void*  *\*/*  **public** **function** onBootstrap($e)  {  *// Register a "render" event, at high priority (so it executes prior*  *// to the view attempting to render)*  $app = $e->getApplication();  $app->getEventManager()->attach('render', **array**($this, 'registerJsonStrategy'), 100);  }  */\*\**  *\* @param \Zend\Mvc\MvcEvent $e The MvcEvent instance*  *\* @return void*  *\*/*  **public** **function** registerJsonStrategy($e)  {  $app = $e->getTarget();  $locator = $app->getServiceManager();  $view = $locator->get('Zend\View\View');  $jsonStrategy = $locator->get('ViewJsonStrategy');  *// Attach strategy, which is a listener aggregate, at high priority*  $view->getEventManager()->attach($jsonStrategy, 100);  }  } |

The above will register the JsonStrategy with the “render” event, such that it executes prior to the PhpRendererStrategy, and thus ensure that a JSON payload is created when requested.

What if you want this to happen only in specific modules, or specific controllers? One way is similar to the last example in the [*previous section on layouts*](http://www.zendframework.com/manual/2.2/en/modules/zend.view.quick-start.html#zend-view-quick-start-usage-layouts), where we detailed changing the layout for a specific module:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | **namespace** Content;  **class** **Module**  {  */\*\**  *\* @param \Zend\Mvc\MvcEvent $e The MvcEvent instance*  *\* @return void*  *\*/*  **public** **function** onBootstrap($e)  {  *// Register a render event*  $app = $e->getParam('application');  $app->getEventManager()->attach('render', **array**($this, 'registerJsonStrategy'), 100);  }  */\*\**  *\* @param \Zend\Mvc\MvcEvent $e The MvcEvent instance*  *\* @return void*  *\*/*  **public** **function** registerJsonStrategy($e)  {  $matches = $e->getRouteMatch();  $controller = $matches->getParam('controller');  **if** (**false** === strpos($controller, \_\_NAMESPACE\_\_)) {  *// not a controller from this module*  **return**;  }  *// Potentially, you could be even more selective at this point, and test*  *// for specific controller classes, and even specific actions or request*  *// methods.*  *// Set the JSON strategy when controllers from this module are selected*  $app = $e->getTarget();  $locator = $app->getServiceManager();  $view = $locator->get('Zend\View\View');  $jsonStrategy = $locator->get('ViewJsonStrategy');  *// Attach strategy, which is a listener aggregate, at high priority*  $view->getEventManager()->attach($jsonStrategy, 100);  }  } |

While the above examples detail using the JsonStrategy, the same could be done for the FeedStrategy.

What if you want to use a custom renderer? Or if your app might allow a combination of JSON, Atom feeds, and HTML? At this point, you’ll need to create your own custom strategies. Below is an example that appropriately loops through the HTTP Accept header, and selects the appropriate Renderer based on what is matched first.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117 | **namespace** Content\View;  **use** Zend\EventManager\EventManagerInterface;  **use** Zend\EventManager\ListenerAggregateInterface;  **use** Zend\Feed\Writer\Feed;  **use** Zend\View\Renderer\FeedRenderer;  **use** Zend\View\Renderer\JsonRenderer;  **use** Zend\View\Renderer\PhpRenderer;  **class** **AcceptStrategy** **implements** ListenerAggregateInterface  {  **protected** $feedRenderer;  **protected** $jsonRenderer;  **protected** $listeners = **array**();  **protected** $phpRenderer;  **public** **function** \_\_construct(  PhpRenderer $phpRenderer,  JsonRenderer $jsonRenderer,  FeedRenderer $feedRenderer  ) {  $this->phpRenderer = $phpRenderer;  $this->jsonRenderer = $jsonRenderer;  $this->feedRenderer = $feedRenderer;  }  **public** **function** attach(EventManagerInterface $events, $priority = **null**)  {  **if** (**null** === $priority) {  $this->listeners[] = $events->attach('renderer', **array**($this, 'selectRenderer'));  $this->listeners[] = $events->attach('response', **array**($this, 'injectResponse'));  } **else** {  $this->listeners[] = $events->attach('renderer', **array**($this, 'selectRenderer'), $priority);  $this->listeners[] = $events->attach('response', **array**($this, 'injectResponse'), $priority);  }  }  **public** **function** detach(EventManagerInterface $events)  {  **foreach** ($this->listeners **as** $index => $listener) {  **if** ($events->detach($listener)) {  unset($this->listeners[$index]);  }  }  }  */\*\**  *\* @param \Zend\Mvc\MvcEvent $e The MvcEvent instance*  *\* @return \Zend\View\Renderer\RendererInterface*  *\*/*  **public** **function** selectRenderer($e)  {  $request = $e->getRequest();  $headers = $request->getHeaders();  *// No Accept header? return PhpRenderer*  **if** (!$headers->has('accept')) {  **return** $this->phpRenderer;  }  $accept = $headers->get('accept');  **foreach** ($accept->getPrioritized() **as** $mediaType) {  **if** (0 === strpos($mediaType, 'application/json')) {  **return** $this->jsonRenderer;  }  **if** (0 === strpos($mediaType, 'application/rss+xml')) {  $this->feedRenderer->setFeedType('rss');  **return** $this->feedRenderer;  }  **if** (0 === strpos($mediaType, 'application/atom+xml')) {  $this->feedRenderer->setFeedType('atom');  **return** $this->feedRenderer;  }  }  *// Nothing matched; return PhpRenderer. Technically, we should probably*  *// return an HTTP 415 Unsupported response.*  **return** $this->phpRenderer;  }  */\*\**  *\* @param \Zend\Mvc\MvcEvent $e The MvcEvent instance*  *\* @return void*  *\*/*  **public** **function** injectResponse($e)  {  $renderer = $e->getRenderer();  $response = $e->getResponse();  $result = $e->getResult();  **if** ($renderer === $this->jsonRenderer) {  *// JSON Renderer; set content-type header*  $headers = $response->getHeaders();  $headers->addHeaderLine('content-type', 'application/json');  } **elseif** ($renderer === $this->feedRenderer) {  *// Feed Renderer; set content-type header, and export the feed if*  *// necessary*  $feedType = $this->feedRenderer->getFeedType();  $headers = $response->getHeaders();  $mediatype = 'application/'  . (('rss' == $feedType) ? 'rss' : 'atom')  . '+xml';  $headers->addHeaderLine('content-type', $mediatype);  *// If the $result is a feed, export it*  **if** ($result instanceof Feed) {  $result = $result->export($feedType);  }  } **elseif** ($renderer !== $this->phpRenderer) {  *// Not a renderer we support, therefor not our strategy. Return*  **return**;  }  *// Inject the content*  $response->setContent($result);  }  } |

This strategy would be registered just as we demonstrated registering the JsonStrategy earlier. You would also need to define DI configuration to ensure the various renderers are injected when you retrieve the strategy from the application’s locator instance.