Recite CMS Container Rule Driver Development Guide

Recite CMS 2.1.3

Recite CMS Container Rule Driver Development Guide

Table of Contents

List of Examples

Chapter 1. Introduction

Containers and container rules are the mechanism by which dynamic content is inserted on a web site created with Recite CMS. Without container rules, a web site will consist only of a series of statis HTML pages.

Every page in Recite CMS can have any number of containers. A container is a area on a page in which container rules can be inserted. Each container can container any number of container rules.

Container rules can output any content or perform any functionality you require. This guide instructs you how to develop your own container rule types to use in conjunction with those that ship with Recite CMS.

The general idea with container rules is that when they are executed, the rule generates various content than can then be displayed with a user-defined template. That is, the container rule shouldn't perform any direct output of its own. This guide will help you achieve this.

In this guide we will create a simple container rule driver called **weather**. This driver asks the user to select their location, then displays the weather for that location.

Chapter 2. Getting Started

Each container rule type is a Recite CMS driver that lives in the `./lib/drivers/containers/rule` directory.

Creating Your Driver File

The driver file is a PHP class with the filename `controller.php` which extends from the `Module_Containers_Driver_Abstract` class.

The class in this file must follow standard Recite CMS driver naming conventions. For example, if your driver is called **weather**, your `controller.php` file would have the path `./lib/drivers/containers/rule/weather/controller.php`.

In this example, the file would define a single class called `Driver_containers_rule_weather_controller`

There are a number of methods you must implement to create your container rule, each of which is covered in this guide. One of these methods is the `__toString()`, which must return a brief description of what the container rule driver does.

Example 2.1. A sample container (`controller.php`)

```
<?php
    class Driver_containers_rule_weather_controller extends Module_Containers_Driver_Abstract
    {
        public function __toString()
        {
            return 'Display the latest weather';
        }

        /* other code will go here */
    }
?>
```

Assigning the Driver to Clients

Once your driver has been created you must assign it to each client you want to be able to use it. This is done using the Recite CMS Administration Site.

Container Rule Templates

When a user adds a new rule to a container, one of the mandatory fields they must complete is the choice of display template. All container rules are rendered on the client's web site using a template.

Every container rule driver automatically defines a new template type that users can choose from when creating a new template. Users can then choose their templates of this type when adding the container rule to their site.

Chapter 3. Processing Container Rule Options

When a user adds your container rule to their web site, the first thing they will need to do is select options for the rule. This includes things such as selecting a display template or deciding how many rows to show (if applicable).

While the processing of some of these options is handled by Recite CMS, any custom options you require must be processed by your driver. To this end, there are three methods available to you.

In addition to these methods, you must also define a display template for the container rule options.

To demonstrate processing a container rule's options we'll create a container rule driver that asks the user to select a location.

Method: getFormDefaults

- `public function getFormDefaults()`
- Returns `array`

This method returns an array of values that are stored with any container rule that uses this driver. If a value isn't mentioned here it will not be stored. The array must have the value name as its key and the default value as the value in the array.

Example 3.1. An example of the `getFormDefaults()` method

```
<?php
    class Driver_containers_rule_weather_controller extends Module_Containers_Driver_Abstract
    {
        // ...

        public function getFormDefaults()
        {
            return array(
                'location' => 'Adelaide'
            );
        }

        // ...
    }
?>
```

Method: getEditorData

- `public function getEditorData()`
- Returns `array`

This method returns an array of data that can be used to help the user select various options. For example, we'll define a list of locations the user can select from. We can then access this data from our display template to output a dropdown list for the user.

Example 3.2. An example of the `getEditorData()` method

```
<?php
    class Driver_containers_rule_weather_controller extends Module_Containers_Driver_Abstract
    {
        // ...

        public function getEditorData()
        {
            return array(
                'locations' => array(
                    'Adelaide', 'Melbourne', 'Sydney', 'Brisbane'
                )
            );
        }

        // ...
    }
?>
```

Method: process

- `public function process(FormProcessor $fp, Zend_Controller_Request_Abstract $request)`

When the user submits their chosen options for the container rule, this method is called. At this point you must process their values to ensure they are valid. This stage will typically involve three parts for each value you want to process:

1. Read a value from the `$request` object.
2. If the value is valid, write it to the form process using `$fp->set('name', $val)`.
3. If the value is invalid, write an error message to the form processor using `$fp->addError('name', 'Some error message to display')`.

Tip

You can learn more about the `Zend_Controller_Request_Abstract` class at http://framework.zend.com/manual/en/zend.controller.request.html.

If no errors are reported, the container rule is saved with the options you write to the form processor. If one or more errors are reported then the user will be shown the error message(s).

The following example demonstrates reading the weather value submitted by the user and acting accordingly. In this example we make use of the data returned by `getEditorData()`

Note

We haven't yet covered how to display an option to the user to select the location, but we'll cover this next.

Example 3.3. Processing a user-submitted value

```
<?php
    class Driver_containers_rule_weather_controller extends Module_Containers_Driver_Abstract
    {
        // ...

        public function process(FormProcessor $fp, Zend_Controller_Request_Abstract $request)
        {
            $location = $request->getPost('location');

            if (in_array($location, $this->getEditorData())) {
                $fp->set('location', $location);
            }
            else {
                $fp->addError('location', 'Please select a valid location');
            }
        }

        // ...
    }
?>
```

Displaying Container Rule Options to the User

Finally we must create a template that displays the options to the user. This template uses Smarty Template Engine syntax, and belongs in the `./templates/index.tpl` file in your container driver directory.

This template should consists of a separate `{field}{/field}` block for each user-definable option. The `{field}` block accepts a number of different arguments, including the title, error key and help value. The error key corresponds to the first argument passed to `addError()` when processing the values. If you set the *`required`* parameter to true, a red asterisk will appear besides the field title.

The data returned from `getEditorData()` is available in the template in the `$fp->data` variable. The following template demonstrates how to use this data so the user can choose a location.

Example 3.4. Displaying container rule options to the user

```
{field title='Location'
       required=true
       error='location'
       help='Select the location to display weather for'}

    <select name="location">
        <option value=""></option>
        {html_options values=$fp->data.locations
                      output=$fp->data.locations
                      selected=$fp->get('location')}
    </select>

{/field}
```

Chapter 4. Rendering Your Container Rule Driver

When a user adds a container rule, they are automatically prompted to select a display template (this is discussed earlier in this guide). In order to render your container rule, the `run()` method is executed (covered below), then the chosen template is rendered.

Method: run

- `public function run(Module_Pages_ClientRenderer_PageRequestTree $pageRequestTree)`
- Throws `Exception`

This method is executed prior to rendering the container rule with the chosen template. This allows you to make any data available to the template as required.

Take the weather container rule driver we've been developing in this guide. In this situation you would determine the weather for the chosen location, then assign it to the *view*. The view is what controls the display of a template. You can assign any data to it, all of which is available from the template.

You can access the view from the `run()` by calling `$this->getView()`. You can then assign any data as required.

In order to access the location value that is stored with the container rule (defined when the user added the container rule), you must access the rule and corresponding parameter. You can firstly calling `$this->getRule()` to retrieve the rule, then `getParam()` to get the value.

The following listing shows an example of this. It generates a fake weather report assigns it to the view (obviously to be of real use it would contact some weather service with the location to get the real data).

Example 4.1. Running a container rule

```
<?php
    class Driver_containers_rule_weather_controller extends Module_Containers_Driver_Abstract
    {
        // ...

        public function run(Module_Pages_ClientRenderer_PageRequestTree $pageRequestTree)
        {
            $rule     = $this->getRule();
            $location = $rule->getParam('location');

            $data = array(
                'location' => $location
                'day'      => 'Monday'
                'temp'     => '30 Celsius'
            );

            $view = $this->getView();
            $view->weather = $data;
        }

        // ...
    }
?>
```

Displaying the Assigned Data

As mentioned previously, the user who added the container rule to their site would also need to create a template with which to display the generated data. For the sake of completeness, here's a template you must use to output the generated weather data.

Example 4.2. Displaying sample weather data

```
<p>
    The weather in {$weather.location|escape}
    on {$weather.day|escape}
    is {$weather.temp|escape}
</p>
```

Creating Sample Templates

One of the features of Recite CMS is that when a user tries to create a new template they can base their template on one of the sample templates that may be available. It is good practice when creating a new container rule driver to define at least one sample template.

You can do this by creating any number of templates in the `./templates/samples` directory within the container rule. Each sample must have the extension `.tpl`. You can then define a description that will appear with the sample in the Recite CMS Control Panel by creating a text file with the same name as the template (but with an extension of `.txt` instead of `.tpl`). This text file should contain a short sentence about what the template does.

For example, if you create a sample called `weather-display.tpl`, you might create a file called `weather-display.txt` which contains the text `Outputs the weather for the chosen location`.

Controlling Container Rule Caching

You can control the cacheability of the data in your template by implementing the `getCacheOptions` method. If defined, this method must return an instance of `Application_Cache_Options`. This class is covered in the main Recite CMS Developer Guide.

Disabling Built-In Template Handling

In some cases you may not want to automatically render a template after a container rule has run. If this is the case, define a method called `usesTemplates` and return false.

Note that doing this will result in the template choice not being shown to the user when creating the container rule, and will also mean that the container rule cannot be cached.

Chapter 5. Paging and Limiting

One of the primary uses of container rules is to display lists of a data. If this is what your custom container rule does, you may want to split up the results into a series of pages.

If you want to enable this functionality, add the following code to your driver. Returning true from second method means the web site builder can choose to allow end-users to override the default limit via a URL parameter.

Example 5.1. Enabling paging and limiting of results

```
<?php
    class Driver_containers_rule_weather_controller extends Module_Containers_Driver_Abstract
    {
        // ...

        public function canBeMultiPaged()
        {
            return true;
        }

        public function limitCanBeSetInUrl()
        {
            return true;
        }

        // ...
    }
?>
```

Adding this code now means when the adds the container rule they can choose whether or not to split up the results. The user can then choose the number of results to display per page, and whether or not the page number and/or limit can be defined by the end-user in the URL.

Recite CMS will then determine the correct values that you can use to fetch your data.

Determining the Current Page Number and Results Limit

If you need to know the current page number when the container rule is running (that is, inside the `run()` method), call the `getPageNumber()` method. Pass to it the page request tree that is passed to `run()`. This will return an integer greater than or equal to 1.

Similarly, you can call `getPageLimit()` to determine the maximum number of results to return. This will return an integer greater than or equal to 0. If the value is zero, then there is no limit.

If required, you can then determine the offset using the formula `(page - 1) * limit`.

The following listing demonstrates this functionality, and how you might use it when building an SQL query.

Example 5.2. Determining the page number and results limit and using them in an SQL query

```
<?php
    class Driver_containers_rule_weather_controller extends Module_Containers_Driver_Abstract
    {
        // ...

        public function run(Module_Pages_ClientRenderer_PageRequestTree $pageRequestTree)
        {
            $page   = $this->getPageNumber($pageRequestTree);
            $limit  = $this->getPageLimit($pageRequestTree);
            $offset = ($page - 1) * $limit;

            $db = Application::GetDb();
            $select = $db->select();
            /* build the select */
            $select->limit($limit, $offset);

            /* perform query and use data */
        }

        // ...
    }
?>
```

Chapter 6. Controlling Results Ordering

Related to splitting results into a series of pages, you may also want to give users the option to determine the order of results. Recite CMS makes this easy to implement for developers.

By enabling this functionality, you can let the web site creator determine the order of data, and you can let them decide if end-users can override this using a URL paramter.

Adding Ordering Capabilities to Your Container Rule

There are three methods you must define to enable the results ordering functionality. Firstly, `usesOrdering()` must return true to enable the ordering options. Next, you must return the available orders from `getSortOrders()`. Finally, you must define the default sort order using `getDefaultSortOrder()`.

Additionally, you can let the web site builder choose if end-users can override the order direction via a URL parameter. To do so, return true from `orderCanBeSetInUrl()`.

The following code demonstrates each of these methods. Note that the `getSortOrders()` method returns array of key/value pairs. The key is the value you use internally in `getDefaultSortOrder` and `run()` (and the value users specify if overriding via the URL), while the value is what is displayed in the container rule options.

Example 6.1. Enabling ordering of results

```
<?php
    class Driver_containers_rule_weather_controller extends Module_Containers_Driver_Abstract
    {
        // ...

        public function usesOrdering()
        {
            return true;
        }

        public function getSortOrders()
        {
            return array(
                'title'  => 'Title (A-Z)',
                'titled' => 'Title (Z-A)'
            );
        }

        public function getDefaultSortOrder()
        {
            return 'titled';
        }

        public function orderCanBeSetInUrl()
        {
            return true;
        }

        // ...
    }
?>
```

Determining the Current Order Direction

Once the ordering functionality has been enabled, you can determine the current page order from within `run()` using the `getSortOrder()`.

You can then use the returned value to determine how to order your data. The following demonstrates how you might do this in SQL.

Example 6.2. Determining the order direction using it in an SQL query

```
<?php
    class Driver_containers_rule_weather_controller extends Module_Containers_Driver_Abstract
    {
        // ...

        public function run(Module_Pages_ClientRenderer_PageRequestTree $pageRequestTree)
        {
            $db = Application::GetDb();
            $select = $db->select();
            /* build the select */

            switch ($this->getSortOrder($pageRequestTree)) {
                case 'title':
                    $select->order('title');
                    break;

                case 'titled':
                default:
                    $select->order('title desc');
            }

            /* perform query and use data */
        }

        // ...
    }
?>
```