Recite CMS Backend Request Driver Development Guide

Recite CMS 2.1.6

Recite CMS Backend Request Driver Development Guide

Table of Contents

List of Examples

Chapter 1. Introduction

In order for a web site powered by Recite CMS to communicate with the Recite CMS installation a backend request driver is used. Common situations where this would arise are as follows:

- When a user submits a form, the forms backend request handler processes the form.
- When an authenticated user wants to log out the users backend request handler accepts this request and logs out the user.
- When a user adds a product to their shopping cart the e-commerce backend request handler deals with this.
- If you want to enable any kind of Ajax interaction between your web site and a custom module, you would use JavaScript to connect a backend request handler.

This guide shows you how to create your own backend request handler.

Chapter 2. Getting Started

Each backend request driver is a Recite CMS driver that lives in the `./lib/drivers/backend/requests` directory.

Creating Your Driver File

The driver file is a PHP class with the filename `driver.php` which extends from the `Module_Backend_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 `driver.php` file would have the path `./lib/drivers/backend/requests/weather/driver.php`.

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

The only mandatory method you must implement is the `__toString()`, which must return a brief description of what the request handler does.

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

```
<?php
    class Driver_backend_requests_weather_controller extends Module_Backend_Driver_Abstract
    {
        public function __toString()
        {
            return 'Return weather data via JSON';
        }

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

Assigning the Driver to Clients

Unlike most other drivers in Recite CMS, backend request drivers are not assigned to clients. Typically these drivers will be related to an existing module (defined by the `getDependentModules()` method). Whether or not a client has access to a backend request driver is based upon their permissions for the dependent modules.

For instance, if we have a module called **weather**, we would define `getDependentModules()` as follows.

Example 2.2. Defining driver dependencies (`driver.php`)

```
<?php
    class Driver_backend_requests_weather_controller extends Module_Backend_Driver_Abstract
    {
        public function getDependentModules()
        {
            return array('weather');
        }

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

Accessing the Driver Through Your Web Site

To access your backend request driver from your Recite CMS, use the URL `/__/drivername`.

Using the weather driver as an example, the URL to access this driver would be `/__/weather`.

In this guide you will see how to handle these requests.

Chapter 3. Handling Requests With run()

When your backend request driver is accessed from the frontend Recite CMS web site, the request is sent through the `run()` method of the driver.

When your driver extends from the `Module_Backend_Driver_Abstract` class, this method is automatically defined for you and makes it relatively easy to create a number of different actions.

Typically you won't need to define `run()` yourself since the built-in one is highly effective (you'll see how in the next chapter). You can skip the rest of this chapter unless you need to define your own `run()`

Method: run

- `public function run($path)`

This method accepts as its only argument the path specified in the request. Only the path remaining after specifying the driver name is included.

For example, if the URL requested was `/__/weather/get/Adelaide`, then `$path` argument contains `get/Adelaide`.

Accessing Form Variables

Additionally, you can access request variables (get/post) by accessing the request with `$this->getRequest()`. This returns an object that inherits from `Zend_Controller_Request_Abstract`.

You can read more about this at `http://framework.zend.com/manual/en/zend.controller.request.html`.

The following example demonstrates reading in a posted variable called `name`.

Example 3.1. Accessing post data (`driver.php`)

```
<?php
    class Driver_backend_requests_weather_controller extends Module_Backend_Driver_Abstract
    {
        public function run($path)
        {
            $request = $this->getRequest();

            $name = $request->getPost('name');
        }

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

Chapter 4. Defining Actions

As mentioned in the previous chapter, you typically won't need to define the `run()` method yourself. The default implementation will use the next value in the requested path.

It will look for a method in the driver called *some*`Action()`.

For example, if the request URL is `/__/weather/get/Adelaide`, the first URL part after the driver name is `get`. This would result in a function called `getAction()` being called.

The remaining path is passed to this method (that is `Adelaide`).

If the request URL is simply `/__/weather`, the default action that is request is `indexAction()`.

If a given method isn't found then an exception is thrown from `run()`, resulting in a 404 file not found response.

To continue with the weather example, the following listing demonstrates how to define an action. Let's assume that the request URL for this would `/__/weather/get` and that it expects a posted value called `city`

Example 4.1. Defining a backend request action (`driver.php`)

```
<?php
    class Driver_backend_requests_weather_controller extends Module_Backend_Driver_Abstract
    {
        public function getDependentModules()
        {
            return array('weather');
        }

        public function indexAction($path)
        {
            $request = $this->getPost();
            $city    = $request->getPost('city');

            // now make use of $city
        }
    }
?>
```

You can have as many actions as you like, each of which can accept different values and return any type of data. We'll look at how to properly respond to a request shortly.

Method: preDispatch and postDispatch

- `public function preDispatch($action)`
- `public function postDispatch($action)`

The `preDispatch()` method is called immediately before an action handler is called, while `postDispatch()` is called immediately after an action handler is called.

The name of the action is passed as the only argument to these methods.

For example, if the requested URL is `/__/weather/get`, then the following calls occur:

1. `$this->preDispatch('get');`

2. `$this->getAction($path);`
3. `$this->postDispatch('get');`

These methods allow you to easily define functionality that is common to all action handlers.

Chapter 5. Completing a Backend Request

Backend requests are typically used to either accept some user-submitted data and process it somehow, or to return data via Ajax. This section shows you how to respond to actions accordingly.

Types of Responses

A request is either going to be an Ajax request or it isn't. If it's not, you either want to return the user back to the web site, or you want to display some kind of message to them.

If it is an Ajax request, there's no use in redirecting the user anywhere since it will be ignored. In this case you either want to send back some data, or you want to do nothing.

Returning to the Main Web Site

By default, once a backend request completes the user will be returned back to the web site.

If you don't want the user returned, call the `setReturnToFrontEnd()` with an argument of `false`. You can call this from the action handler, from `preDispatch()`, or from `postDispatch()`.

Setting the Return URL

There are two ways to define the URL to return to. By default, when trying to return the user, Recite CMS will check for a posted form value called `return`. If this is specified then this is the URL that the user is returned to.

Alternatively, you can define the `getReturnUrl()` method. This method must return a string. The returned string is the location the user is referred to.

If the return URL cannot be determined, Recite CMS will look for a referring URL for the backend request. If one is found, then that is the return URL.

As a last-ditch effort, the root URL of your web site is used as the return URL if no other URL can be determined.

Responding to Ajax Requests

The typical way to respond to an Ajax request is to send back JSON-encoded data. You can do this easily by calling the `$this->sendJson()` method.

This method accepts as its only argument the data you would like encoded as JSON data.

The following listing demonstrates this functionality. It returns JSON data in all cases (regardless of whether the request was an Ajax request or not).

Example 5.1. Return JSON data from a request (`driver.php`)

```
<?php
    class Driver_backend_requests_weather_controller extends Module_Backend_Driver_Abstract
    {
        public function getDependentModules()
        {
            return array('weather');
        }

        public function indexAction($path)
        {
            $request = $this->getPost();
            $city    = $request->getPost('city');

            $weather = array(
                'city' => $city,
                'temp' => '35 Celsius'
            );

            $this->sendJson($weather);
        }
    }
?>
```

In some cases you may want to send JSON-encoded data only if the request was an Ajax request. This is useful when making your web site degrade gracefully for browsers that don't support JavaScript.

You can check the `isXmlHttpRequest()` method on the request to check for this.

Example 5.2. Return JSON data only for Ajax requests (`driver.php`)

```
<?php
    class Driver_backend_requests_weather_controller extends Module_Backend_Driver_Abstract
    {
        public function getDependentModules()
        {
            return array('weather');
        }

        public function indexAction($path)
        {
            $request = $this->getPost();
            $city    = $request->getPost('city');

            $weather = array(
                'city' => $city,
                'temp' => '35 Celsius'
            );

            if ($request->isXmlHttpRequest()) {
                $this->sendJson($weather);
            }
            else {
                // don't do anything - the user will be returned
            }
        }
    }
?>
```

Outputting Content Directly

If you want to output content directly (rather than sending JSON or returned the user back to the web site), you can do so.

Firstly, you must set the driver to not return back to the web site (covered earlier in this chapter).

Next, you must access the HTTP response using `$this->getResponse()`. This returns an object that extends from `Zend_Controller_Response_Abstract`. Refer to `http://framework.zend.com/manual/en/zend.controller.response.html` for more details.

Call the `setBody()` method on the response object to set the body to return. You can also set any headers as required (this can be useful for sending back files, such as generated PDF files).