Recite CMS Control Panel Widget Development Guide (BETA GUIDE ONLY)

Recite CMS 2.1.8

Recite CMS Control Panel Widget Development Guide (BETA GUIDE ONLY)

Table of Contents

List of Examples

Chapter 1. Introduction

The Recite 2 Control Panel uses a widget-based system, meaning all aspects of managing content on a Recite web site involves interacting with a series of different widgets.

A Control Panel user can easily manage their widgets. Each user can have any number of tabs; a tab has one or more columns; each column can have any number of widgets; widgets can be added, removed or repositioned on-demand.

Widgets are written using a combination of PHP, JavaScript, HTML. Sometimes you will also use custom CSS. JavaScript is heavily used.

Your widgets can interact with other parts of Recite as you please:

- They can interact with modules you develop. This is the typical scenario for using custom modules - if you develop a custom module you'll need a way to manage its data.
- They can interact with existing modules. For instance, you could write your own file upload widget if you didn't want to use the default Recite File Import widget.
- They can interact with third-party services. For instance, you could write a widget that displays statistical data from Google Analytics.
- They can interact with other widgets. For instance, you might want to add a new widget when a certain action occurs, or you might want to tell an existing widget about something that just happened on your custom widget.

This guide teaches you how to develop your own widgets for Recite.

Getting Started

To develop custom widgets you will need an understanding of PHP, HTML and JavaScript. Recite makes use of the jQuery JavaScript framework.

Additionally, you will need an Integrated Development Environment (IDE) or text editor with which to write code, as well as a working copy of Recite 2.

Chapter 2. Creating a Widget

In this section I'll show you how to create a widget. This will only be a basic widget that demonstrates the most basic development details; more advanced topics are covered later in this guide.

In this example we will create a widget called **custom_example**.

Directory Structure

All Control Panel widgets belong within the `widgets` directory in the Recite library (that is, in `./lib/widgets`).

You must create a new sub-directory for your widget. This can either be within one of the existing directories or you can create your own.

We are creating a widget called **custom_example**, so we create the directory `./lib/widgets/custom/example`. All of our widget files belong this directory. No other widgets can then belong within this widget.

We will create this widget based on the `Application_Cp_Widget_Module_Abstract` class, which dictates the directory structure of the widget.

This directory then has the following structure:

```
./lib/widgets/custom/example
  | - files/               Holds files that can be loaded in the Control Panel
  |    | - js/
  |    |    | - index.js   Holds JavaScript controller file for "index" action
  |    | - css/
  |    |    | - index.css  Holds a stylesheet to help render the template
  | - templates/           Holds templates used to render the widget
  |    | - index.tpl       Template for the "index" action
  | - controller.php       Main controller for widget
```

We'll now cover how each of these files should be structure.

Creating PHP Controller File

This is the main file required for a widget. In this file we define a new PHP class for the widget, which extends from the `Application_Cp_Widget_Module_Abstract` class.

Our widget class must be named `CpWidget_`*`widget_name`*`_controller`. So in our case the class will be called `CpWidget_custom_example_controller`.

The skeleton code for this class is as follows:

```
<?php
    class CpWidget_custom_example_controller extends Application_Cp_Widget_Module_Abstract
    {
    }
?>
```

There are several methods that you must implement in this class so it can be used in the Control Panel:

- `__toString()`. This returns a string that is the title of your widget.

- `getWidgetSize()`. This returns an indication of how much space on the user's screen your widget needs to be used effectively. The Control Panel will aim to give the widget its required amount of space when added to a user's dashboard.

  This method can return one of the following constant values: `self::SIZE_SMALL`, `self::SIZE_MEDIUM`, or `self::SIZE_LARGE`.

- `getCategories`. This returns an array, each of which element is a string. The values it returns will appear in the widget browser when the user tries to add a new widget.

- `getDescription()`. This returns a brief description of what the widget is used for. It will appear in the widget browser when a user views the widget.

- `getDependentModules()`. This returns an array, each of which element is the string name of any Recite modules this widget relies on. If the module doesn't exist in Recite or the client the user belongs to doesn't have access to the module, the widget cannot be added and it won't appear in the wiget browser.

- `indexAction()`. This method is executed when the widget is loaded in the Control Panel. Here you can put any logic required for displaying the widget. After this method has been run the corresponding `./templates/index.tpl` template will be rendered. You can also optionally define the JavaScript controller file for this action in `./files/js/index.js`

Note

In addition to `indexAction()`, you can also define other actions that can be called from your widget. To keep things simple we'll only use a single action for now.

Here's a more complete example of the `controller.php` file.

Example 2.1. Sample widget controller file (`controller.php`)

```
<?php
    class CpWidget_custom_sample_controller
        extends Application_Cp_Widget_Module_Abstract
    {
        public function __toString()
        {
            return 'Sample Widget';
        }

        public function getWidgetSize()
        {
            return self::SIZE_SMALL;
        }

        public function getCategories()
        {
            return array('Custom');
        }

        public function getDescription()
        {
            return translate(
                'This widget is used to demonstrate how to create Recite widgets.'
            );
        }

        public function getDependentModules()
        {
            return array('pages');
        }

        public function indexAction()
        {
            // perform custom logic here

            // get the view so you can assign data to it
            // before the index.tpl template is rendered

            $view = $this->getView();
            $view->someData = 'Some data!';
        }
    }
?>
```

At this stage you can now add the widget to the Control Panel. However, until you create a view template nothing useful will display.

Creating a View Template

The next step is to create a view template for the `indexAction()` method. This template is a text file that belongs in `./templates/index.tpl` in your widget directory. This template must be written using HTML and Smarty Template Engine markup.

Technically you can use whatever markup you like in this file, but to use the standard Control Panel display all your markup should be within `{widget}` Smarty tag. This is a built-in plug-in used to help creating widget templates.

If you want a row of buttons at the top of your widget (beneath the wiget title and control buttons), use the `{widget_buttonpane header=true}` Smarty tag.

Likewise, if you want a row of buttons at the bottom of your widget use the `{widget_buttonpane footer=true}` Smarty tag.

To add a button, use the `{button}` Smarty tag.

Any content that lies between button panes (or even if there are no button panes) should lie within `{widget_content}` tags.

Below is a sample template. It makes uses of the `$someData` variable we assigned in the `controller.php` file.

Example 2.2. Sample widget view template (`index.tpl`)

```
{widget}
    {widget_buttonpane header=true}
        {button name='foo'}Show Alert{/button}
    {/widget_buttonpane}

    {widget_content}
        someData variable: {$someData|escape}
    {/widget_content}
{/widget}
```

Now when you add your widget you will see some more useful information.

Creating a JavaScript Controller File

In the previous template example we created a button. In order to make this button do anything useful we need a JavaScript controller file.

This file belongs in the `./files/js/index.js`, and defines a new JavaScript object that defines certain methods.

The object this JavaScript file defines must be called `CpWidget_widget_name_index`. In our example this would be `CpWidget_custom_example_index`.

This object can define the following methods, each of which is passed the widget's DOM element as its only argument:

- `init()`. This is called when a widget is created prior to it appearing on the user's dashboard. It is also called when a widget is refreshed.
- `postShow()`. This is called after the widget is displayed on the dashboard.
- `destroy()`. This is called when a widget is removed; when a new tab is loaded; just prior to a widget being refreshed.

Typically you will only need to define the `init()` method, although sometimes you will also need the `destroy()` method.

Below is an example of how this file should look. In this example we bind the `click` event to the button we created. When the button is clicked a dialog box will appear.

> **Note**
>
> We'll cover the Recite_Tabs and Recite_Dialog user-interface classes later in this guide.

Example 2.3. Sample JavaScript controller file (`index.js`)

```
var CpWidget_custom_example_index = {

    init : function(widget)
    {
        Recite_Tabs.Bind(
            widget,
            widget.find('button[name=foo]'),
            'click',
            function(e) {
                e.preventDefault();

                Recite_Dialog.Alert({
                    msg : 'Button was clicked!'
                });
            }
        );
    }
};
```

In this example we bind the `click` event using the `Recite_Tabs.Bind()` method, rather than calling `bind()` directly on the button element. We do this so the event is automatically unbound when the widget is destroyed. The alternative is to manually unbind the event in the `destroy()` method.

Creating a Widget CSS File

You can style content within a widget as required by creating a CSS file. As noted earlier in this chapter, it is stored in the `./files/css` directory of the widget. The name of the CSS file corresponds to the action it is being loaded for. Typically this will simply be `index.css`.

When creating your CSS it is possible to override page-wide styles. Typically this is not desirable, so you need to restrict your CSS selectors to the given widget. You can do this by qualifying all selectors with a CSS class that is the same name as the widget.

For instance, you could override all `div` elements in the **CpWidget_custom_example** widget by defining a CSS rule of `.CpWidget_custom_example div { }`.

> **Note**
>
> The CSS class name does not include the action name.

Summary

In this chapter we've covered the most important parts of creating a widget. From here we'll build on what we can do in widgets, including communicating with Recite modules and using built-in user interface components.

Chapter 3. Event Handling

The Recite Control Panel is an event-driven system. In other words, widgets will respond to certain things happening.

Let's use the File Import widget as an example:

1. User selects a file to upload then begins the upload. *Trigger an event that an "Uploading" status message should be shown.*
2. File continues to upload. *Listen for an event that indicates how much has been uploaded and update the display accordingly.*
3. File completes uploading. *Trigger an event that the "Uploading" status message should no longer be shown.*
4. Receive confirmation from server that file is uploaded. *"File Browser" widget listens for this event. Refreshes its display so the newly uploaded file is displayed.*

You can trigger your own events and listen for events accordingly.

Naming of Events

Events are named in the format *eventName.nameSpace*. The namespace is determined by what triggers the event. Events triggered by modules are named module_*moduleName*.

Let's return to the previous file upload example to demonstrate this. When a file is uploaded, the server sends back the following event: `filecreated.module_assets`.

The Recite Delegate Element

In the Control Panel there is a hidden special element called the Recite Delegate which all events go through (this is aside from normal events that occur on DOM elements).

This element is accessible using the `Recite.Delegate` variable in your JavaScript controllers.

Triggering Events

To trigger an event on the delegate, call the `trigger()` method. The first argument is the name of the event, while the second is any custom data you want to include with the element.

Let's use the file browser widget as an example. When a file is clicked, we trigger the `fileselected.module_assets` event. We pass to this event the internal ID of the file that was selected.

The code to achieve this is as follows:

```
Recite.Delegate.trigger(
    'fileselected.module_assets',
    {
        id : fileId
    }
);
```

Listening For Delegate Events

To listen for any events in your widget, you can use the `bind()` method on the delegate element. Let's listen for the `fileselected.module_assets` event we listened for:

```
Recite.Delegate.bind(
    'fileselected.module_assets',
    function(e, memo) {
        Recite_Dialog.Alert({
            msg : 'File with ID ' + memo.id + ' was selected'
        });
    }
);
```

Note, however, that there is a fundamental flaw in this code: we need to unbind this listener when the widget is destroyed, and this code doesn't allow us to do this easily.

You could call `Recite.Delegate.unbind('fileselected.module_assets')`, however, this would mean every other widget listening for this event would also be unbound.

To deal with this, you must either pass the function as the second argument to `unbind()` (meaning you can't use an anonymous function like in our example), or you can just use the `Recite_Tabs.Delegate()` method to originally bind the event.

```
Recite_Tabs.Delegate(
    widget,
    'fileselected.module_assets',
    function(e, memo) {
        Recite_Dialog.Alert({
            msg : 'File with ID ' + memo.id + ' was selected'
        });
    }
);
```

Binding the event in this manner means it will automatically be unbound when the widget is destroyed.

Note

You must pass the widget as the first argument to `Delegate()`.

Listening For Other Events

You can listen for normal events (such as `click`) on normal events, just like we did in the sample `index.js` earlier in this guide.

Once again, you must unbind the event when you're done, so to help with this we have the `Recite_Tabs.Bind()` method. The first argument is the widget; the second argument is the element (or list of elements) to bind the event to; the third argument is the event to bind; the final argument is the event handler function.

Here's the code we used to bind an event to the button we created when creating our sample widget.

```
Recite_Tabs.Bind(
    widget,
    widget.find('button[name=foo]'),
    'click',
    function(e) {
        e.preventDefault();
```

```
        Recite_Dialog.Alert({
            msg : 'Button was clicked!'
        });
    }
);
```

Automatically Triggered Events

Many events will be automatically triggered by the response to Ajax calls. For example -- as we covered earlier -- when a file is uploaded the server will send back a `filecreated.module_assets` event.

Refer to the Ajax Requests chapter for details on how send Ajax requests and how to send responses containing events.

Chapter 4. Status Messages

When developing Ajax-powered applications it is important to keep users informed when background actions are occurring. If they don't know that something is happening when they expect something to be happening they might grow impatient or give up.

To keep users informed, Recite provides a mechanism for displaying status messages. There are two types of status messages that can be displayed:

- Loading messages. These are displayed when a background action is occurring. This includes processing a form, loading data, or any other action is occurring that the user may need to wait upon.
- Status updates. These are displayed after some action has occurred. For instance, if you upload a file with the File Import widget a "File uploaded" message is displayed to the user so they know the action has completed.

Many status messages are triggered automatically from responses to Ajax requests. For more information on how send status messages from Ajax requests refer to the Ajax Requests chapter.

Triggering Loading Messages

To display a loading message, use the following code:

```
Recite.Delegate.trigger('loadstart.status');
```

This will display a status message that says `Loading...`. You can display a custom message by passing an object with a string called `msg` as the second argument to `trigger()`.

For example, to display the message `Doing something...`, you would use the following code:

```
Recite.Delegate.trigger('loadstart.status', { msg : 'Doing something...' });
```

By default the status message will be attached to the main browser window. If you want to attach it to a different element you can pass the parent DOM element in the `parent` element of the second argument.

Hiding Loading Messages

Once your loading action has complete you should hide the status message so the user knows loading is complete. Use the following code to hide the loading message:

```
Recite.Delegate.trigger('loadend.status');
```

If you passed a custom parent element when triggering the status message, you must pass that element once again when hiding the loading message.

Triggering Status Updates

To display a status message you can do so in a similar fashion to displaying a loading message. The difference is that you trigger the `infostart.status` event, you must include the message to display, and you must specify the type of message.

Each message type will display in a different colour to indicate to the user whether an operation was successful or if some error occurred. The available types are as follows:

- `Recite_Status.Types.INFO`. An informational message.
- `Recite_Status.Types.SUCCESS`. Used to indicate success.
- `Recite_Status.Types.NOTICE`. Used to indicate something worthwhile happened to the user. Typically this status type is used when an item is deleted.
- `Recite_Status.Types.WARN`. Used to display some kind of warning.
- `Recite_Status.Types.ERR`. Used to indicate an error occurred.
- `Recite_Status.Types.DEBUG`. Used to display a debugging message.

For example to trigger a message that indicates success, you might use the following code:

```
Recite.Delegate.trigger(
   'info.status',
   {
       msg  : 'Something good happened!',
       type : Recite_Status.Types.SUCCESS
   }
);
```

Hiding Status Messages

Recite will take care of hiding status messages automatically. If you really must hide a status message, you can trigger the `infoend.status` event.

```
Recite.Delegate.trigger('infoend.status');
```

Chapter 5. Ajax Requests

This chapter is yet to be written, but the following code demonstrates a controller and controller action that set a status message and a custom event.

```
<?php
    class Mymodule_SomeController extends Application_Module_Controller
    {
        public function indexAction()
        {
            $this->addStatusMessage(
                'Some status message',
                Application_Site_Response_StatusMessage::STATUS_SUCCESS
            );

            $this->addEvent('myevent.module_mymodule')
                 ->addMemo('val1', 'Some value');
        }
    }
?>
```

Chapter 6. Form Processing

Chapter 7. Widget Management

Chapter 8. JavaScript API

This section documents the various JavaScript components that are available to widget developers.

Component Recite

This components contains a number of useful helper functions core to the operation of Recite CMS widgets. There are additional utility methods that can be used in Recite_Util.

Method: Recite.GetUrl

`<static> {String} Recite.GetUrl(target)`

Get a real Recite CMS URL based on a string in module:controller:action syntax

```
var url = Recite.GetUrl('assets:asset:upload');
```

- Parameters:
  - `String target` - The URL in module:controller:action syntax
- Returns:
  - `String` The real URL that can be used for links or Ajax requests

Method: Recite.HasCss

`<static> {Bool} Recite.HasCss(path)`

Check if the CSS file with the given path has been loaded

```
if (Recite.HasCss('/path/to/styles.css')) { ... }
```

- Parameters:
  - `String path` - The path of the CSS script to check for
- Returns:
  - `Bool` Returns true if the CSS file has previously been loaded, false if not.

Method: Recite.LoadCss

`<static> Recite.LoadCss(path)`

Loads a CSS file. If the file has already been loaded it will not be loaded again.

```
Recite.LoadCss('path/to/styles.css');
```

- Parameters:
  - `String path` - The path of the CSS file to load

- Returns: `Void`

Method: Recite.LoadScript

`<static> Recite.LoadScript(path, fn)`

Load a JavaScript script based on the specified path. If the script has already dynamically been loaded with this function it will not be loaded again but the callback function will be called again.

```
Recite.LoadScript('/path/to/script.js, function() {
    alert('script is loaded!');
});
```

- Parameters:
    - `String path` - The path to the script to load
    - `Function fn` - *Optional* - The function to call when the script has loaded. Accepts no arguments
- Returns: `Void`

Component Recite_Ajax

This components contains methods for performing Ajax requests in the Recite CMS Control Panel.

Method: Recite_Ajax.Ajax

`<static> Recite_Ajax.Ajax(url, method, data, fnSuccess, fnError)`

Perform an Ajax request. This will automatically interpret any returned events or messages from PHP action handlers (unless you use your own success and/or error callbacks

```
Recite_Ajax.Ajax(
    Recite.GetUrl('assets:asset:import'),
    'post',
    {
        'foo' : 'bar'
    }
);
```

- Parameters:
    - `String url` - The URL to send the request to
    - `String method` - *Optional* - The request method to use (get or post)
    - `Object data` - *Optional* - Any additional data to send with the request
    - `Function fnSuccess` - *Optional* - Callback function when request completes (request may be HTTP error)
    - `Function fnError` - *Optional* - Callback function when request does not complete (e.g. if network down)
- Returns: `Void`

Method: Recite_Ajax.Error

`<static> Recite_Ajax.Error(xhr, textStatus, errorThrown)`

This is the callback method for a failed Ajax request. It automatically processes any included events and status messages. You can chain this callback to your own Ajax handler by simply called Recite_Ajax.Success() at the end of your handler (remembering to include the original parameters). This will automatically display a status message indicating that an error occurred.

```
Recite_Ajax.Post(
    Recite.GetUrl('assets:asset:import'),
    {
        'foo' : 'bar'
    },
    null,
    function(xhr, textStatus, errorThrown)
    {
        // do something, then chain to error
        Recite_Ajax.Error(xhr, textStatus, errorThrown);
    }
);
```

- Parameters:
    - `XMLHttpRequest xhr` - The transport object
    - `String textStatus` - The text status message
    - `Mixed errorThrown` - The error that occurred
- Returns: `Void`

Method: Recite_Ajax.Get

`<static> Recite_Ajax.Get(url, data, fnSuccess, fnError)`

Wrapper function for calling Recite_Ajax.Ajax() as a get request

```
Recite_Ajax.Get(
    Recite.GetUrl('assets:asset:import')
);
```

- Parameters:
    - `String url` - The URL to send the request to
    - `Object data` - *Optional* - Any additional data to send with the request
    - `Function fnSuccess` - *Optional* - Callback function when request completes (request may be HTTP error)
    - `Function fnError` - *Optional* - Callback function when request does not complete (e.g. if network down)
- Returns: `Void`
- See:

- Recite_Ajax.Ajax

Method: Recite_Ajax.Post

`<static> Recite_Ajax.Post(url, data, fnSuccess, fnError)`

Wrapper function for calling Recite_Ajax.Ajax() as a post request

```
Recite_Ajax.Post(
    Recite.GetUrl('assets:asset:import'),
    {
        'foo' : 'bar'
    }
);
```

- Parameters:
    - `String url` - The URL to send the request to
    - `Object data` - *Optional* - Any additional data to send with the request
    - `Function fnSuccess` - *Optional* - Callback function when request completes (request may be HTTP error)
    - `Function fnError` - *Optional* - Callback function when request does not complete (e.g. if network down)
- Returns: `Void`
- See:
    - Recite_Ajax.Ajax

Method: Recite_Ajax.Success

`<static> Recite_Ajax.Success(data)`

This is the callback method for a successful Ajax request. It automatically processes any included events and status messages. You can chain this callback to your own Ajax handler by simply called Recite_Ajax.Success() at the end of your handler (remembering to include the original response JSON data)

```
Recite_Ajax.Post(
    Recite.GetUrl('assets:asset:import'),
    {
        'foo' : 'bar'
    },
    function(data)
    {
        // do something, then chain to success
        Recite_Ajax.Success(data);
    }
);
```

- Parameters:
    - `String data` - The data returned from an Ajax request

- Returns: `Void`

Component Recite_Dialog

Methods used for managing dialog boxes in the Recite CMS Control Panel

Method: Recite_Dialog.Alert

`<static> Recite_Dialog.Alert(alertOptions)`

Show an alert box with a static message. Alert boxes are simply information boxes with a single Ok button.

- Parameters:
  - `Object alertOptions` - The options used for displaying the dialog
  - `String alertOptions.msg` - The message to display in the alert box.
  - `String alertOptions.title` - *Optional, Default:* `"Alert"` - The text to use for the dialog title
  - `String alertOptions.okText` - *Optional, Default:* `"Ok"` - The text to use for the OK button
  - `String|Integer alertOptions.width` - *Optional, Default:* `"500"` - The width in pixels of the dialog
  - `String|Integer alertOptions.height` - *Optional, Default:* `"auto"` - The height in pixels of the dialog
- Returns: `Void`

Method: Recite_Dialog.Confirm

`<static> Recite_Dialog.Confirm(confirmOptions)`

Show a confirmation dialog using a static message. A confirmation dialog is a dialog that typically shows two buttons: "Ok" and "Cancel".

- Parameters:
  - `Object confirmOptions` - The options used for displaying the dialog
  - `String confirmOptions.url` - *Optional* - The URL to post to when OK is clicked (if custom onOk callback is not specified).
  - `String confirmOptions.msg` - The message to display in the confirmation box.
  - `Object confirmOptions.data` - *Optional* - Additional data to post when OK is clicked (if custom onOk callback is not specified).
  - `String confirmOptions.method` - *Optional, Default:* `"GET"` - Whether the request should be GET or POST
  - `String confirmOptions.okText` - *Optional, Default:* `"Ok"` - The text to use for the OK button
  - `String confirmOptions.cancelText` - *Optional, Default:* `"Cancel"` - The text to use for the cancel button
  - `String confirmOptions.title` - *Optional, Default:* `"Please Confirm"` - The text to use for the dialog title

- `Function confirmOptions.onOk` - *Optional* - Function to execute when OK is clicked. Accepts dialog as its only argument. Dialog is not automatically closed if this is specified.
- `Function confirmOptions.onCancel` - *Optional* - Function to execute when cancel is clicked. Accepts dialog as its only argument. Dialog is not automatically closed if this is specified.

- Returns: `Void`

Method: Recite_Dialog.ConfirmFromUrl

```
<static> Recite_Dialog.ConfirmFromUrl(confirmOptions)
```

Show a confirmation dialog using content loaded via AjaxA confirmation dialog is a dialog that typically shows two buttons: "Ok" and "Cancel".

- Parameters:
  - `Object confirmOptions` - The options used for displaying the dialog
  - `String confirmOptions.url` - *Optional* - The URL to post to when OK is clicked (if custom onOk callback is not specified).
  - `Object confirmOptions.data` - *Optional* - Additional data to post when OK is clicked (if custom onOk callback is not specified).
  - `String confirmOptions.method` - *Optional, Default: `"GET"`* - Whether the request should be GET or POST
  - `String confirmOptions.msgUrl` - The URL to retrieve the dialog message from
  - `Object confirmOptions.msgData` - *Optional* - Additional data to use in request for dialog message. If request is get and includes query parameters, this should not be specified.
  - `String confirmOptions.okText` - *Optional, Default: `"Ok"`* - The text to use for the OK button
  - `String confirmOptions.cancelText` - *Optional, Default: `"Cancel"`* - The text to use for the cancel button
  - `String confirmOptions.title` - *Optional, Default: `"Please Confirm"`* - The text to use for the dialog title
  - `Function confirmOptions.onOk` - *Optional* - Function to execute when OK is clicked. Accepts dialog as its only argument. Dialog is not automatically closed if this is specified.
  - `Function confirmOptions.onCancel` - *Optional* - Function to execute when cancel is clicked. Accepts dialog as its only argument. Dialog is not automatically closed if this is specified.
- Returns: `Void`

Method: Recite_Dialog.FromAjax

```
<static> Recite_Dialog.FromAjax(url, data, method)
```

Fetch a dialog box and display it. The URL the dialog is fetch from must build a dialog using the PHP class Application_Cp_Ui_Dialog (or a sub-class) and return it accordingly.

- Parameters:
  - `String url` - The URL to retrieve the dialog from

- `Object data` - *Optional* - Any additional data to include in the Ajax request
- `String method` - *Optional, Default:* *`"get"`* - The request method to use (get or post)

- Returns: `Void`

Method: Recite_Dialog.SelectorFromJSON

```
<static> Recite_Dialog.SelectorFromJSON(selectorOptions)
```

Show a dialog box with a single dropdown box. Options are retrieved from the supplied URL, then the user must select an option. When they click Ok then selected option is passed to the supplied callback. The selector must return data generated by a PHP sub-class of Components_DriverSelector_Options_Abstract. Users can either select an option or cancel the dialog.

- Parameters:
  - `Object selectorOptions` - The options used to generate the dialog
  - `String selectorOptions.url` - The URL to retrieve the options from.
  - `String selectorOptions.emptyMsg` - *Optional* - The message to display if no options are found
  - `String selectorOptions.emptyTitle` - *Optional, Default:* *`"No options found"`* - The title to use if no options are found
  - `String selectorOptions.msg` - *Optional* - The message to display above the dropdown
  - `String selectorOptions.okText` - *Optional, Default:* *`"Ok"`* - The text to display in the "Ok" button
  - `String selectorOptions.cancelText` - *Optional, Default:* *`"Cancel"`* - The text to display in the "Cancel" button
  - `Function selectorOptions.select` - The function that is called when an option is selected. The value of the selected option is passed as the only argument
- Returns: `Void`

Component Recite_DragDrop

Methods for managing drag/drop between Recite CMS Control Panel widgets. This works in conjunction with jQuery UI's draggable components

Method: Recite_DragDrop.BuildDragContainer

```
<static> {Element} Recite_DragDrop.BuildDragContainer(dragNode, options)
```

Build a drag container that will be displayed when one or more items are being dragged

- Parameters:
  - `Element dragNode` - The element being dragged
  - `Object options` - Additional options for building the drag container
  - `String[] options.items` - The identifiers of one or more items being dragged

- `String options.singular` - *Optional, Default: `"item"`* - A singular name for the type of item being dragged
- `String options.plural` - *Optional, Default: `"items"`* - A name for the type of item being dragged
- `String options.type` - A name to identify the type of node begin dragged. This is used so droppables know whether or not to accept the item.

- Returns:
  - `Element` The created DOM node that will be displayed as the drag element

Method: Recite_DragDrop.GetDragInfo

```
<static> {Object} Recite_DragDrop.GetDragInfo(dragNode)
```

This element returns information about what was dragged from a node that has just been dragged.

- Parameters:
  - `Element dragNode` - The node that was just dragged. This should be the same element passed to BuildDragContainer
- Returns:
  - `Object` Information about the drag. The type is in the type property and the item IDs are in the items property/

Component Recite_Form

Methods used to manage forms in the Recite CMS Control Panel

Method: Recite_Form.ClearErrors

```
<static> Recite_Form.ClearErrors(form)
```

Clear all errors previously populated with the PopulateErrors method

- Parameters:
  - `Element form` - The DOM element to remove form errors from
- Returns: `Void`
- See:
  - Recite_Form.PopulateErrors

Method: Recite_Form.CreateWysiwyg

```
<static> Recite_Form.CreateWysiwyg(options)
```

Create a WYSIWYG editor dialog. You can add custom buttons to it and handle any buttons as required. If you use the default buttons then an Ok and a Cancel button are shown.

- Parameters:
  - `Object options` - The options used to build the WYSIWYG dialog

- `String options.title` - *Optional, Default: `"WYSIWYG"`* - The title of the dialog
- `Object options.buttons` - *Optional* - Each element is indexed by the button label and the value is the callback function. The dialog DOM element is passed as the first argument and the contents of the WYSIWYG editor is passed as the second argument. The dialog will not be automatically closed when these buttons are clicked.
- `String options.okText` - *Optional, Default: `"Ok"`* - If not manually specifying buttons this is the text that will appear on the OK button
- `Function options.ok` - *Optional* - If not manually specifying buttons this is the function to call when the OK button is clicked. The WYSIWYG editor data is passed as the only argument to this method. The dialog is automatically closed.
- `String options.cancelText` - *Optional, Default: `"Cancel"`* - If not manually specifying buttons this is the text that will appear on the cancel button
- `Function options.cancel` - *Optional* - If not manually specifying buttons this is the function to call when the cancel button is clicked. The WYSIWYG editor data is passed as the only argument to this method. The dialog is automatically closed.

- Returns: `Void`

Method: Recite_Form.PopulateErrors

```
<static> Recite_Form.PopulateErrors(form, formErrors)
```

Populate a form with errors return from an Ajax request. Errors are populated based on their name, which must match up with the name passed to {form_error} in the form template

- Parameters:
  - `Element form` - The DOM element of the form
  - `String[] formErrors` - The errors returned from the form.submitted event
- Returns: `Void`

Method: Recite_Form.SerializeToObject

```
<static> {Object} Recite_Form.SerializeToObject(form)
```

Serialize a form into a normal object. This can then be passed as data to the Recite_Ajax.Ajax method

- Parameters:
  - `Element form` - The form DOM element to serialize
- Returns:
  - `Object` The serialized form values
- See:
  - Recite_Ajax.Ajax

Method: Recite_Form.ShowFormSuccess

```
<static> Recite_Form.ShowFormSuccess(form, msg)
```

Indicate that a form was successfully submitted. This message will be shown where the "global" form error is normally display

- Parameters:
  - `Element form` - The form DOM element that the success message should be shown within
  - `String msg` - The message to display
- Returns: `Void`

Method: Recite_Form.SubmitViaAjax

```
<static> Recite_Form.SubmitViaAjax(form, fnSuccess, fnError)
```

Submit a form using Ajax

- Parameters:
  - `Element form` - The DOM element of the form to submit
  - `Function fnSuccess` - *Optional* - The function to execute if the form was all valid. Accepts the form DOM element and the submitted.form event memo entry as its two arguments
  - `Function fnError` - *Optional* - The function to execute if the form was all not valid. Accepts the form DOM element and the submitted.form event memo entry as its two arguments
- Returns: `Void`

Component Recite_Tree

Functions related to using the JavaScript tree component in Recite CMS

Method: Recite_Tree.GetDefaultOptions

```
<static> {Object} Recite_Tree.GetDefaultOptions()
```

Get default options to use for building a tree. This should be used as a starting point when a tree needs to be loaded.

- Returns:
  - `Object`

Method: Recite_Tree.GetId

```
<static> {String|Integer} Recite_Tree.GetId(id)
```

Retrieve the ID number from a tree-friendly internal ID

- Parameters:
  - `String id` - The tree-friendly ID to extract the real ID from
- Throws:
  - `String`

    If the node isn't in a valid format

- Returns:
    - `String|Integer` The real ID

Method: Recite_Tree.SetId

`<static> {String} Recite_Tree.SetId(widgetId, id)`

Get a tree-friendly ID based on the widget ID and the real ID

- Parameters:
    - `Integer widgetId` - The ID of the widget this ID is being used for
    - `String|Integer id` - The real ID
- Returns:
    - `String` The tree-friendly ID

Component Recite_Util

Various utility methods for Recite CMS

Method: Recite_Util.GetHighlightMessage

`<static> {Element} Recite_Util.GetHighlightMessage(msg)`

Get a DOM element that displays a highlighted message based using the input string

- Parameters:
    - `String msg` - The message to highlight
- Returns:
    - `Element` The highlighted element

Method: Recite_Util.UrlGenerator

`<static> Recite_Util.UrlGenerator(src, dst, separator)`

Automatically write a friendly URL-part into a text input based on the value of another input. If the URL input is manually modified it will stop auto-generating.

- Parameters:
    - `Element src` - The HTML input the URL part is generated from
    - `Element dst` - The HTML input the URL part is written to
    - `String separator` - *Optional, Default:* `"-"` - The string to separate words in the generated URL
- Returns: `Void`

Method: Recite_Util.Urlize

`<static> {String} Recite_Util.Urlize(str, separator)`

Make a string URL friendly. The resultant string will have only letters and numbers, and words will be separated by the specified separator.

- Parameters:
    - `String str` - The string to urlize
    - `String separator` - *Optional, Default:* `"-"` - The word separator
- Returns:
    - `String` The urlized string

Appendix A. Triggered Events

Every Recite CMS module triggers various events in various situations. This appendix lists these events and data provided with them.

Module: Ads

The events in this section are triggered when ad management operations occur.

Zone Operations

The following events are triggered for zone management.

- `zonecreated.module_ads`. Triggered when a zone is created.
- `zonemodified.module_ads`. Triggered when an existing zone is modified.
- `zonedeleted.module_ads`. Triggered when a zone is deleted.

Each of these events send back the following data.

- `id`. Internal ID of the zone.
- `title`. Title of the zone.

Campaign Operations

The following events are triggered for campaign management.

- `campaigncreated.module_ads`. Triggered when a campaign is created.
- `campaignmodified.module_ads`. Triggered when an existing campaign is modified.
- `campaigndeleted.module_ads`. Triggered when a campaign is deleted.

Each of these events send back the following data.

- `id`. Internal ID of the campaign.
- `title`. Title of the campaign.

Linking Campaigns to Zones

The following events are triggered in relation to linking zones with campaigns.

- `campaignlinkedtozone.module_ads`. Triggered when a campaign is linked to a zone.
    - `zone_id`. Internal ID of the zone.
    - `campaign_id`. Internal ID of the campaign.
- `campaignunlinkedfromzone.module_ads`. Triggered when a campaign is unlinked from a zone.
    - `zone_id`. Internal ID of the zone.
    - `campaign_id`. Internal ID of the campaign.

Banner Operations

The following events are triggered for campaign management.

- `bannercreated.module_ads`. Triggered when a banner is created.
- `bannermodified.module_ads`. Triggered when an existing banner is modified.
- `bannerdeleted.module_ads`. Triggered when a banner is deleted.

Each of these events send back the following data.

- `id`. Internal ID of the banner.
- `campaign_id`. Internal campaign ID the banner belongs to.
- `title`. Title of the banner.

Module: Assets

Events in this section apply to file management operations.

File Operations

The following events apply to file management. Each of these events relate to a single file.

- `filecreated.module_assets`. Triggered when a new file is created.
- `filemodified.module_assets`. Triggered when an existing file is updated.
- `filedeleted.module_assets`. Triggered when an existing file is deleted.
- `fileextracted.module_assets`. Triggered when an archive (such as a zip file) is extracted.
- `filerestored.module_assets`. Triggered when a file is restored to an older version.

The following data is included with each of these events.

- `id`. Internal ID of the file.
- `folder_id`. Folder ID of the file.
- `filename`. Current filename of the file.

In addition, the following event is also triggered:

- `fileselected.module_assets`. Triggered when a file is selected in the Control Panel.
    - `id`. The ID of the file that was selected.

Folder Operations

The following events apply to folder management. Each of these events relate to a single folder.

- `foldercreated.module_assets`. Triggered when a new folder is created.
- `foldermodified.module_assets`. Triggered when an existing folder is updated.
- `folderdeleted.module_assets`. Triggered when an existing folder is deleted.

The following data is included with each of these events.

- `id`. Internal ID of the folder.
- `parent_id`. Folder ID of the folder's parent folder.
- `name`. Current name of the folder.

In addition, the following event is also triggered:

- `folderselected.module_assets`. Triggered when a folder is selected in the Control Panel.
    - `id`. The ID of the folder that was selected.

Bulk Operations

The following events are sent back when bulk operations occur.

- `filebulkmoved.module_assets`. Triggered when files are bulk moved.
    - `folder_ids`. An array of folder IDs that were affected by the move.
- `filebulkdeleted.module_assets`. Triggered when one or more files are bulk deleted.
    - `file`. This is an array where each entry corresponds to a moved file. This element contains `id`, `name` and `folder_id` values.
- `folderbulkdeleted.module_assets`. Triggered when one or more folders are bulk deleted.
    - `folder`. This is an array where each entry corresponds to a moved folder. This element contains `id`, `name` and `parent_id` values.
- `bulkdeleted.module_assets`. Triggered when files and/or folders are bulk deleted. This will be triggered with at least one of the previous two events.
    - `count`. The total number of files/folders deleted.

Linking Files to Other Content

The following events relate to linking files to other content.

- `linkedfilesupdated.module_assets`. Triggered when some content has its linked files updated.
    - `driver`. The name of the connector linking the file(s) to the content.
    - `linked_id`. The internal ID of the linked item.
    - `title`. A descriptive title for the linked item.

Module: Assets Mirrors

- .
    - .
    - .
    - .
- .

- .
- .
- .
- .
  - .
  - .
  - .
- .
  - .
  - .
  - .
- .
  - .
  - .
  - .
- .
  - .
  - .
  - .

Module: Calendar

- .
  - .
  - .
  - .
- .
  - .
  - .
  - .
- .
  - .
  - .

        - .
    - .
        - .
        - .
        - .
    - .
        - .
        - .
        - .
    - .
        - .
        - .
        - .

Module: Categories

- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .

        - .
        - .
        - .
    - .
        - .
        - .
        - .

Module: Comments

- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .

- .

Module: E-commerce

- .
  - .
  - .
  - .
- .
  - .
  - .
  - .
- .
  - .
  - .
  - .
- .
  - .
  - .
  - .
- .
  - .
  - .
  - .
- .
  - .
  - .
  - .

Module: Feeds

- .
  - .
  - .
  - .

- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .

Module: Forms

- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .

        - .
    - .
        - .
        - .
        - .
    - .
        - .
        - .
        - .
    - .
        - .
        - .
        - .

Module: Listings

- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .

- .
- .
- .

- .
- .
- .
- .

Module: Mailing Lists

- .
- .
- .
- .

- .
- .
- .
- .

- .
- .
- .
- .

- .
- .
- .
- .

- .
- .
- .
- .

- .
- .
- .

- .
- .

- .

Module: Pages

- .
  - .
  - .
  - .
- .
  - .
  - .
  - .
- .
  - .
  - .
  - .
- .
  - .
  - .
  - .
- .
  - .
  - .
  - .
- .
  - .
  - .
  - .

Module: Search

- .
  - .
  - .
  - .

- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .

Module: Templates

- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .

        - .
    - .
        - .
        - .
        - .
    - .
        - .
        - .
        - .
    - .
        - .
        - .
        - .

Module: Users

- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .
    - .
    - .
    - .
- .

- .
- .
- .
- .
  - .
  - .
  - .