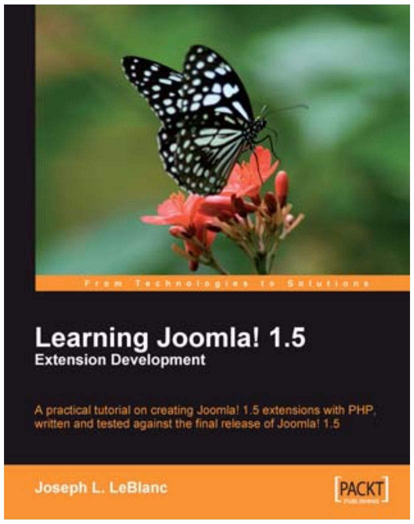
Learning Joomla! 1.5 Extension Development

Joseph L. LeBlanc


From Technologies to Solutions
Learning Joomla! 1.5
Extension Development
A practical tutorial on creating Joomla! 1.5 extensions with PHP, written and tested against the final release of Joomla! 1.5
Joseph L. LeBlanc
PACKT

Chapter No. 8

"Using JavaScript Effects"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.8 "Using JavaScript Effects"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Joseph L. LeBlanc started with computers at a very young age. His independent education gave him the flexibility to experiment with and learn computer science. Joseph holds a bachelor's degree in Management Information Systems from Oral Roberts University.

Joseph is a freelance Joomla! extension developer. He released a component tutorial in May 2004, which was later translated into French, Polish, and Russian. Work samples and open-source extensions are available at www.jlleblanc.com. In addition to freelancing, he is an active member of the Washington, DC tech community and Joomla! Bug Squad.

I would like to thank the following people for making this book possible:

- Packt Publishing, for giving me the opportunity to author this work.
- Everyone who bought the first edition of this book, for offering both praise and critiques. By speaking up, you helped make this edition a reality.
- The Joomla! team, for developing some of the best software in the world.
- The DC PHP community, for showing me different ways people are solving similar problems.
- Steve and Sue Meeks, for their flexibility with my schedule during the writing process and for giving Joomla! a shot.
- Everyone who has downloaded and used my open-source components.
- My professors, for taking me on the Journey of a Byte and showing me how to write effectively.
- Mom and Dad, for teaching me how to learn.

Learning Joomla! 1.5 Extension Development

Although Joomla! has all of the basic content management tools you need to build a website, it is also designed to also run custom-built extensions written in PHP. This book steps through working examples of PHP code written to work seamlessly in Joomla!. Topics covered in this book include libraries for generating user interface elements, database table classes, Model-View-Controller design, configuration panels, the use of JavaScript libraries, and URL routing. After all of the code has been written, it is bundled in `.zip` files, ready to be installed by Joomla! site webmasters.

What This Book Covers

Chapter 1 gives you an overview of how Joomla! works. The example project used throughout the book is also introduced. The three types of extensions (components, modules, and plug-ins) are covered, along with a description of how they work together.

Chapter 2 begins with the development of the component used in the project. Initial entries are made in the database and toolbars are built for the backend. The general file structure of Joomla! is also introduced.

Chapter 3 walks through the creation of the backend interface for creating, editing, and deleting records in the project through the Model-View-Controller design pattern. Database table classes are also introduced.

Chapter 4 builds a frontend interface for listing and viewing records. Additionally, code to generate and interpret search engine friendly links is covered. The project is also expanded slightly, as a commenting feature is added.

Chapter 5 takes a closer look at the methods provided by the JTable, JHTML, and JUser classes. The JTable class allows you to manage a list of ordered records, while JHTML helps generate common HTML elements. Also, the concept of checking out records when JTable and JUser are used together is introduced.

Chapter 6 introduces a module used to list records on every page of the site. The module takes advantage of layouts, where the same data can be formatted differently depending on how the code is called. Some of the code is also separated out into a helper class so that the main code generating the module stays simple.

Chapter 7 continues development of the component, adding elements from the JTHML class that make the component blend in with the rest of the Joomla! interface. Controls over the publishing of records are introduced, as well as an interface for removing offensive comments. More toolbars are added, and multiple controllers are introduced.

Chapter 8 shows how to add many common JavaScript effects to your extensions. This chapter also explains how to create a Google Map and interact with it using MooTools. Finally, a way of using jQuery alongside MooTools is covered.

Chapter 9 develops three plug-ins. The first plug-in finds the names of records in the database and turns them in to links to those records. A second plug-in displays a short summary of the record when certain code is added to content articles. Finally, another plug-in is designed so that records are pulled up alongside Joomla! content searches.

Chapter 10 adds configuration parameters to the components, modules, and plug-ins. These are handled through XML and generate a predictable interface in the backend for setting options. Retrieving the value of these parameters is standardized through built-in functions.

Chapter 11 adds links and functionality to the component where users can email pages to their friends. It also prepares the user interface for internationalization and does a partial translation into French. Additionally, the chapter provides a solution for handling uploaded files.

Chapter 12 expands the XML files used for parameters and adds a listing of all the files and folders in each extension. Once this file is compressed along with the rest of the code into a ZIP archive, it is ready to be installed on another copy of Joomla! without any programmer intervention. Custom installation scripts and SQL code are also added to the component.

8
Using JavaScript Effects

As our critics continue to write reviews and moderate comments, we will take some time to look into options for improving the user interface and adding new functions. Modern websites use JavaScript-driven effects to aid navigation, reduce on-screen clutter, and provide interactive features that are not possible with static HTML. Joomla! has several built-in elements that you can use without writing a single line of JavaScript. The MooTools framework powers many elements seen throughout the Joomla! backend UI; these can be reused in both the frontend and the backend. We will learn to use JavaScript effects through these topics:

- Modal boxes
- Tool tips
- Sliding panes
- Customizing Google Maps
- Using jQuery

Modal boxes

There will be some times when you will want to highlight a piece of information without making visitors load a completely separate webpage. The "lightbox" effect is now frequently used across the web as a way of doing this. When you click on a link where this effect is applied, the webpage is grayed out and a small window with the required content floats on top of the page. Users are prevented from clicking elsewhere on the page until the window is closed. This window is referred to as a modal box.

To demonstrate the use of modal boxes (along with the other examples in this chapter), we will create a component separate from "Restaurant Reviews". Go to the `components` folder of your Joomla! installation and create a folder named `com_js`. Within this folder, create a file named `js.php` with the following code:

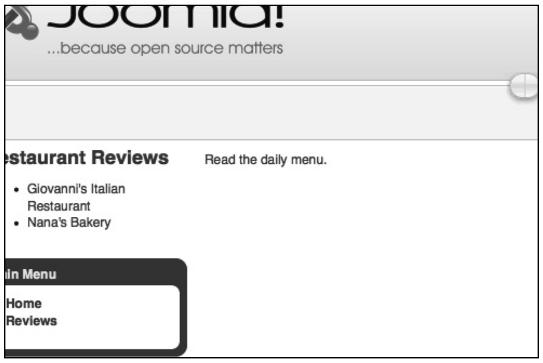
```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );

jimport('joomla.application.component.controller');

class JsController extends JController
{
    function modalBox()
    {
        JHTML::_('behavior.modal', 'a.popup');

    ?>
    <a href="index.php?option=com_js&amp;task=insideModal&amp;
    format=raw" class="popup">Read the daily menu.</a>
        <?php
    }

    function insideModal()
    {
    ?>
        <h1>Today's Menu</h1>
        <ul>
            <li>Crispy chicken nuggets with ginger dressing</li>
            <li>Swordfish in bean curd sauce</li>
            <li>Stir-fried vegetables over white rice</li>
        </ul>
        <?php
    }
}

$controller = new JsController();
$controller->execute(JRequest::getCmd('task'));
```

After checking to make sure that the request to execute `js.php` is coming from within Joomla!, the controller code is loaded from the Joomla! framework. `JsController` is then declared as an extension of `JController`. After the class is defined, `$controller` is set as a new instance of `JsController` and the `execute()` member function is called, with the current task passed in.

The controller contains two functions—`modalBox()` and `insideModal()`. The `modalBox()` function is intended to be displayed as a standard page in Joomla!, while `insideModal()` has the HTML content that will fill the modal window. Inside `modalBox()`, there is a call to `JHTML::_('behavior.modal', 'a.popup');`. Calling `JHTML::_('behavior.modal')` tells the Joomla! framework to load the JavaScript necessary for powering modal boxes. The parameter `'a.popup'` tells the JavaScript powering the modal box that we want to add this effect to all anchor tags that have a class of `popup`.

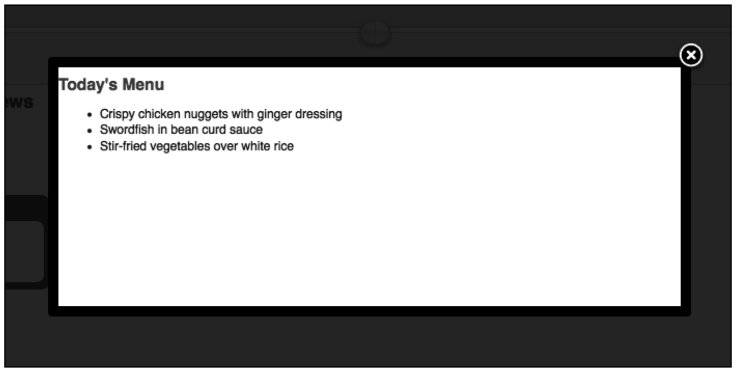
An anchor tag is output after the `JHMTL::_()` call. This link points back to the `com_js` component and sets the task to `insideModal`. Unlike most links in Joomla!, this one specifies the desired format of the output. Setting `format` to `raw` ensures that Joomla! does not attempt to load the template or any of the modules when generating the output; we only want the HTML intended for the window to be loaded.

If you go to `index.php?option=com_js&task=modalBox` now, you should see a screen similar to the following one:



Read the daily menu.

When you click on the **Read the daily menu** link, a window will appear on the screen like this:



Today's Menu

- Crispy chicken nuggets with ginger dressing
- Swordfish in bean curd sauce
- Stir-fried vegetables over white rice

Notice that although JavaScript powers this effect, we did not write a single line of it. The call to the `JHTML` class did this for us. To see the JavaScript that `JHTML` generated, use your browser's **View Source** function. Within the `<head>` tags of the HTML source, you should see a portion of code similar to the following:

```
<link rel="stylesheet" href="/media/system/css/modal.css" type=
"text/css" />
<script type="text/javascript" src=
"/media/system/js/mootools.js"></script>
<script type="text/javascript" src=
"/media/system/js/modal.js"></script>
<script type="text/javascript">
        window.addEvent('domready', function(){
            SqueezeBox.initialize({});

            $$('a.popup').each(function(el) {
                el.addEvent('click', function(e) {
                    new Event(e).stop();
                    SqueezeBox.fromElement(el);
                });
            });
        });
</script>
```

Joomla! makes frequent use of the MooTools JavaScript framework, and the code powering the modal box is no exception. After loading some CSS for formatting the appearance of the box, MooTools itself is loaded. The modal box JavaScript is loaded once MooTools is available. These files are static and are distributed with Joomla!.

The third `<script>` tag includes the JavaScript dynamically generated by Joomla! for our specific modal box. All of the code is enclosed within a call to `window.addEvent()`. This JavaScript function is added to the `window` object by MooTools and allows us to add event handlers. In this case, we are adding a handler to the `domready` event so that the code waits until the DOM is fully loaded.

On the highlighted line of the code, notice the string `a.popup`. This is the same selector we passed in as the second parameter of `JHTML::_()`. We can pass in any desired class selector here, but in this case we only want to attach the event to anchor tags with a class of `popup`. The rest of the code cycles through all of the `a.popup` elements in the HTML document and applies the MooTools `SqueezeBox` plug-in to each of them.

Configuring the modal box

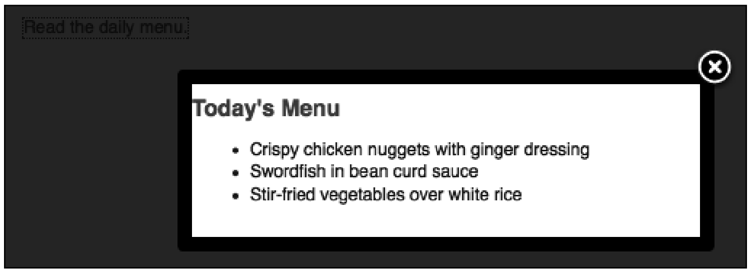
Although the current use of `JHTML::_()` pulls in the desired behavior, we would like to make some adjustments. The default window for the modal is rather large for the content we are displaying. A way of controlling the height and width would be helpful.

Fortunately, there is a way of configuring the modal box. The call to `JHTML::_(' behavior.modal')` takes a multidimensional array containing settings as an additional parameter. Make the highlighted adjustments to the `modalBox()` function in the `/components/com_js/js.php` file:

```
function modalBox()
{
    $params = array(
        'size' => array(
            'x' => 350,
            'y' => 250
            )
        );

    JHTML::_('behavior.modal', 'a.popup', $params);

    ?>
    <a href="index.php?option=com_js&amp;task=insideModal&amp;format=
     raw" class="popup">Read the daily menu.</a>
    <?php
}
```

The `$params` array has one element for the `size` setting. This element is an array with the width and height values we want to use; these are represented as `x` and `y` respectively. This specific array above sets the width to 350 pixels and the height to 250 pixels. The `$params` array is now loaded with our desired configuration, so we pass it in as the third parameter of `JHTML::_()`.

Save `js.php` and reload `index.php?option=com_js&task=modalBox` in the browser. After clicking on the **Read the daily menu** link, the modal box should appear as shown below as shown below:



Today's Menu

- Crispy chicken nuggets with ginger dressing
- Swordfish in bean curd sauce
- Stir-fried vegetables over white rice

The JavaScript generated by Joomla! has also changed. Use your browser's **View Source** function to pull up the HTML code of the page. Then look for the block of `<script>` declarations in the `<head>` section where the modal box is defined. The code should look similar to this:

```
<link rel="stylesheet" href="/media/system/css/modal.css" type=
"text/css" />
<script type="text/javascript" src=
"/media/system/js/mootools.js"></script>
<script type="text/javascript" src=
"/media/system/js/modal.js"></script>
<script type="text/javascript">

        window.addEvent('domready', function() {

            SqueezeBox.initialize({ size: { x: 350, y: 250}});

            $$('a.popup').each(function(el) {
                el.addEvent('click', function(e) {
                    new Event(e).stop();
                    SqueezeBox.fromElement(el);
                });
            });
        });
</script>
```

The generated JavaScript is almost exactly the same as it was before, except for the highlighted line. In the first example, above an empty object was passed into `SqueezeBox.initialize()`. In this example, the `$params` array we passed into `JHTML::_()` has been transformed by Joomla! into a JavaScript object. This object is now passed into `SqueezeBox.initialize()` to configure the `SqueezeBox` plug-in.

The raw format and MVC

In this example, two functions in the controller are used—one to generate the main page and one to fill the modal box with HTML content. However, complex components will be written using views as well. Instead of using a separate controller function for displaying the modal box HTML, a view can be used.

To start, create a `views` folder in the existing `/components/com_js` folder. Within this new folder, create another folder named `modalcontent`. For a typical view, you would create a file in this folder named `view.html.php`. This view is a little different; we will be setting the format to `raw` so we can get only the HTML code for the contents and not an entire Joomla! page with the template. To handle the `raw` format, create the file `view.raw.php` instead. Fill this file with the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );

jimport( 'joomla.application.component.view');

class JsViewModalcontent extends JView
{
}
```

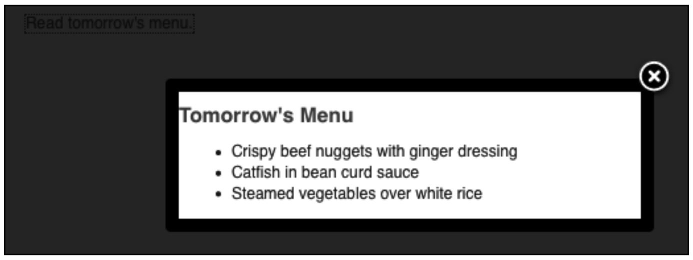
This code might seem a little odd at first glance. Each view in Joomla! must be represented by an object that is an extension of `JView`. By default, if the `display()` function is not overridden in the child object, `JView::display()` will be called instead. This function will load `tmpl/default.php` (unless the layout parameter is set to something other than `default`). Because this example does not load anything from the database or act on the custom variables passed to it, it is desirable to allow Joomla! to execute the default `JView::display()` function. The file `default.php` must exist in the `views/modalcontent/tmpl` folder, though. Create the folder `tmpl`, and then create a file named `default.php`, and fill it with the following code:

```
<?php defined( '_JEXEC' ) or die( 'Restricted access' ); ?>
<h1>Tomorrow's Menu</h1>
<ul>
    <li>Crispy beef nuggets with ginger dressing</li>
    <li>Catfish in bean curd sauce</li>
    <li>Steamed vegetables over white rice</li>
</ul>
```

As with other layout files in Joomla! views, and with all `.php` files, we first check to make sure the call is coming from within Joomla!. Then, the static HTML content is output. To demonstrate the use of this view instead of a controller function, create a slightly modified version of the `modalBox()`, in `/components/com_js/js.php`, and name it `modalBoxMVC()`:

```
function modalBoxMVC()
{
    $params = array(
        'size' => array(
            'x' => 350,
            'y' => 250
            )
        );

    JHTML::_('behavior.modal', 'a.popup', $params);

    ?>
    <a href="index.php?option=com_js&amp;view=modalcontent&amp;
     format=raw" class="popup">Read tomorrow's menu.</a>
    <?php
}
```

Save all of the open files, and then load `index.php?option=com_js&task=modalBoxMVC` in your browser and click on the **Read tomorrow's menu.** link. Your screen should look like the example shown below:



Tomorrow's Menu

- Crispy beef nuggets with ginger dressing
- Catfish in bean curd sauce
- Steamed vegetables over white rice

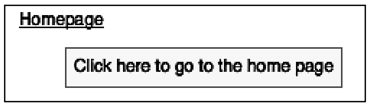
Tool tips

Another effect used across the web is the tool tip. Tool tips are used to add messages that are displayed near the mouse pointer when you hover over specific elements on the screen. This is helpful for adding definitions of things without taking up space on the page. Adding a tool tip in Joomla! is even simpler than adding modal boxes—you don't need to make a separate call to display the content. In the `/components/com_js/js.php` file, add the following function to the controller:

```
function toolTipTest()
{
    JHTML::_('behavior.tooltip');
    ?>
        <span class="hasTip" title="Click here to go to the
        home page"><a href="index.php">Homepage</a></span>
    <?php
}
```

As with the modal box, we call `JHTML::_()` to pull in the MooTools framework. This time, `behavior.tooltip` is passed as the parameter. After this, an anchor tag is output, to create a link. This anchor tag is wrapped in a `<span>` that has two attributes—`class` and `title`. The class attribute is set to `hasTip`; this is the default class the tool tip JavaScript looks for when setting up the effect. The `title` attribute determines the text to be displayed when you move the mouse over the link.

After saving `js.php`, load `index.php?option=com_js&task=toolTipTest` and move your mouse over the **Homepage** link. The tool tip should appear, similar to this example:



Click here to go to the home page

If you want to use a class other than `hasTip` for your elements with tool tips, you can override it in `JHTML::_()`. For example, to use the class `mytip` insted, make the highlighted modifications to the function `toolTipTest()` in `/components/com_js/js.php` file:

```
function toolTipTest()
{
    JHTML::_('behavior.tooltip', '.mytip');
    ?>
        <span class="mytip" title="Click here to go to the
        home page"><a href="index.php">Homepage</a></span>
    <?php
}
```

When you save `js.php` and reload `index.php?option=com_js&task=toolTipTest`, the functionality will be the same as before, but the class will be `mytip`.

Sliding panes

Throughout the backend of Joomla!, there are several screens containing sliding panes full of options. This is done to save space on the screen, and to make specific settings easier to find. The CSS that creates the "tabbed" visual effect is automatically included in the backend. To use the siding panes, you only need to bring in the JavaScript and make some calls to a `JPane` object.

To test the panes, create a component in the backend at `/administrator/components/com_js`. Add the file `js.php` to this folder, and fill it with the following code:

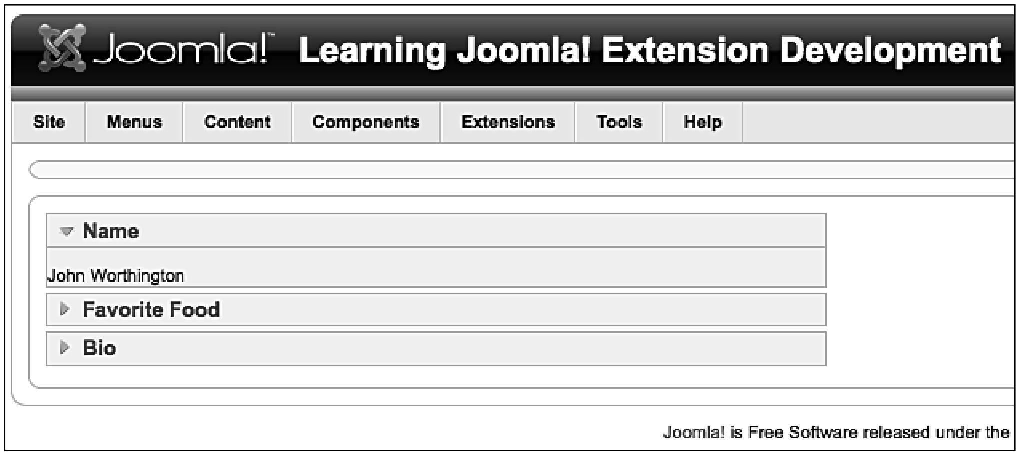
```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );

jimport('joomla.application.component.controller');

class JsController extends JController
{
    function showPanes()
    {
        jimport('joomla.html.pane');

        ?>
        <div class="col width-45">
        <?php

        $pane = &JPane::getInstance('sliders');

        echo $pane->startPane('menu-pane');
        echo $pane->startPanel('Name', 'info-name');
```

```
            echo '<p>John Worthington</p>';
            echo $pane->endPanel();
            echo $pane->startPanel('Favorite Food', 'info-food');
            echo '<p>Pad Thai</p>';
            echo $pane->endPanel();
            echo $pane->startPanel('Bio', 'info-bio');
            echo '<p>John began criticizing food in kindergarden and has
             not stopped since. His accomplishments include earning "2005
             Critic of the Year" from Digest Digest.</p>';
            echo $pane->endPanel();
            echo $pane->endPane();

            ?>
            </div>
            <?php
        }
    }
    $controller = new JsController();
    $controller->execute(JRequest::getCmd('task'));
```

As with the frontend of the component, the backend checks to make sure the file is called within Joomla!, and then defines a controller and executes it. The `showPanes()` function starts with a call to `jimport('joomla.html.pane');` to load the `JPane` class. The `<div>` tag has classes of `col` and `width-45`; these are defined in the backend CSS to display a column that takes 45% of the available screen width.

Within the `<div>`, `$pane` is set using the `getInstance()` member function of `JPane`. The string `'sliders'` is passed into `getInstance()` as `JPane` is capable of handling other similar effects using the same function calls. Once `$pane` is set, the member functions `startPane()`, `endPane()`, `startPanel()`, and `endPanel()` are called in sequence. The `startPane()` and `endPane()` functions are used to enclose all of the slider's panels. `startPane()` also requires a parameter that defines the HTML `id` used on the `<div>` that encloses the panels. Each panel is then defined using `startPanel()`, followed by the content for the panel, and ending with `endPanel()`. The `startPanel()` function accepts a title to be displayed as the first parameter and an HTML `id` for the panel as the second.

To see the `JPane` slider in action, save `js.php` and load `administrator/index.php?option=com_js&task=showPanes` in your browser. You should get the following screen:


Joomla! Learning Joomla! Extension Development
Site
Menus
Content
Components
Extensions
Tools
Help
Name
John Worthington
Favorite Food
Bio
Joomla! is Free Software released under the

As you click on a different heading, the panel for that heading will expand as the previous one slides up. No matter what you click, one and only one pane is always visible. If you want to allow every panel to be closed, make the highlighted modification within `showPanes()`:

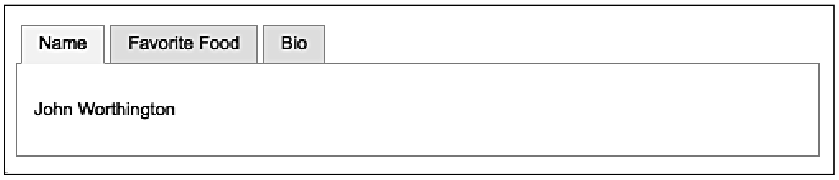
```
<?php

$pane = &JPane::getInstance('sliders',
 array('allowAllClose' => true));

echo $pane->startPane('menu-pane');
```

As with the modal box, the array that is passed in gets transformed into a JavaScript object that is output when Joomla! writes the JavaScript powering the panes. This array defines the `'allowAllClose'` setting as `true`.

Through `JPane::getInstance()`, the sliding panes can quickly be turned into a tabbed interface. Change `'sliders'` to `'tabs'` by making the highlighted change:

```
<?php

$pane = &JPane::getInstance('tabs');

echo $pane->startPane('menu-pane');
```

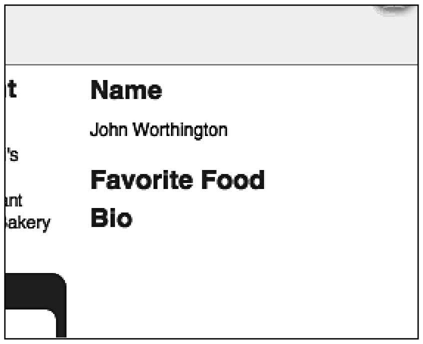
After saving `js.php`, reload `administrator/index.php?option=com_js&task=showPanes` in your browser. You should now have a tabbed interface that looks like the following example:



The sliders can be used in the frontend as well as the backend of Joomla!. To use the sliders in the frontend, open the file `/components/com_js/js.php`, and add the following function to the controller:

```
function frontendPanes()
{
    jimport('joomla.html.pane');

    $pane = &JPane::getInstance('sliders');

    echo $pane->startPane('menu-pane');
    echo $pane->startPanel('Name', 'info-name');
    echo '<p>John Worthington</p>';
    echo $pane->endPanel();
    echo $pane->startPanel('Favorite Food', 'info-food');
    echo '<p>Pad Thai</p>';
    echo $pane->endPanel();
    echo $pane->startPanel('Bio', 'info-bio');
    echo '<p>John began criticizing food in kindergarden and has not
     stopped since. His accomplishments include earning "2005 Critic
     of the Year" from Digest Digest.</p>';
    echo $pane->endPanel();
    echo $pane->endPane();
}
```

The code works in exactly the same way as it does in the backend, only we are not wrapping the sliders in a `<div>` this time. The frontend does not have the CSS that the backend does, so the display will be different, but the functionality will be the same. Save `js.php`, and load `index.php?option=com_js&task=frontendPanes` in your browser. Your screen should now look like this:



Name

John Worthington

Favorite Food

Bio

Customizing Google Maps

Although modal boxes and sliding panes are useful, MooTools can help with other JavaScript tasks. Google Maps has a comprehensive API for interacting with maps on your website. MooTools can be used to load the Google Maps engine at the correct time. It can also act as a bridge between the map and other HTML elements on your site.

To get started, you will first need to get an API key to use Google Maps on your domain. You can sign up for a free key at `http://code.google.com/apis/maps/signup.html`. Even if you are working on your local computer, you still need the key. For instance, if the base URL of your Joomla installation is `http://localhost/joomla`, you will enter `localhost` as the domain for your API key.

Once you have an API key ready, create the file `basicmap.js` in `/components/com_js`, and fill it with the following code:

```
window.addEvent('domready', function() {
    if (GBrowserIsCompatible()) {

        var map = new GMap2($('map_canvas'));
        map.setCenter(new GLatLng(38.89, -77.04), 12);

        window.onunload=function(){
            GUnload();
        };

    }
});
```

The entire script is wrapped within a call to the MooTools-specific `addEvent()` member function of `window`, just like the `SqueezeBox` script for the modal box, earlier. Because we want this code to execute once the DOM is ready, the first parameter is the event name `'domready'`. The second parameter is an anonymous function containing our code.

What does the call to `function()` do?

Using `function()` in JavaScript is a way of creating an anonymous function. This way, you can create functions that are used in only one place (such as event handlers) without cluttering the namespace with a needless function name. Also, the code within the anonymous function operates within its own scope; this is referred to as a closure. Closures are very frequently used in modern JavaScript frameworks, for event handling and other distinct tasks.

Once inside of the function, `GBrowserIsCompatible()` is used to determine if the browser is capable of running Google Maps. If it is, a new instance of `GMap2()` is declared and bound to the HTML element that has an id of `'map_canvas'` and is stored into `map`. The call to `$('map_canvas')` is a MooTools shortcut for `document.GetElementById()`.

Next, the `setCenter()` member function of `map` is called to tell Google Maps where to center the map and how far to zoom in. The first parameter is a `GLatLng()` object, which is used to set the specific latitude and longitude of the map's center. The other parameter determines the zoom level, which is set to 12 in this case. Finally, the `window.onunload` event is set to a function that calls `GUnload()`. When the user navigates away from the page, this function removes Google Maps from memory, to prevent memory leaks.

With our JavaScript in place, it is now time to add a function to the controller in `/components/com_js/js.php` that will load it along with some HTML. Add the following `basicMap()` function to this file:

```
function basicMap()
{
    $key = 'DoNotUseThisKeyGetOneFromCodeDotGoogleDotCom';

    JHTML::_('behavior.mootools');

    $document =& JFactory::getDocument();
    $document->addScript('http://maps.google.com/maps?file=api&v=
        2&key=' . $key);
    $document->addScript(
        JURI::base() . 'components/com_js/basicmap.js');

    ?>
    <div id="map_canvas" style="width: 500px; height: 300px"></div>
    <?php
}
```

The `basicMap()` function starts off by setting `$key` to the API key received from Google. You should replace this value with the one you receive at `http://code.google.com/apis/maps/signup.html`. Next, `JHTML::_('behavior.mootools');` is called to load MooTools into the `<head>` tag of the HTML document. This is followed by getting a reference to the current document object through the `getDocument()` member function of `JFactory`. The `addScript()` member function is called twice—once to load in the Google Maps API (using our key), then again to load our `basicmap.js` script. (The Google Maps API calls in all of the functions and class definitions beginning with a capital 'G'.)

Finally, a `<div>` with an id of `'map_canvas'` is sent to the browser. Once this function is in place and `js.php` has been saved, load `index.php?option=com_js&task=basicMap` in the browser. Your map should look like this:

Foxhall
Adams Morgan
Burleith
Dupont Circle
Logan Circle
Eckington
Georgetown
K St NW
Washington
Independence Ave SW
Arlington
Arlington National Cemetery
East Potomac Park
Anacostia Park
Anacostia
Anacostia Naval Station
Trinidad
Kingman Park
Langston Golf Course
Anacostia River Park
Gateway
Randle Highlands
Fort Stanton Park
Army Navy Country Club
Glencarlyn Park
Map data ©2008 Tele Atlas - Terms of Use

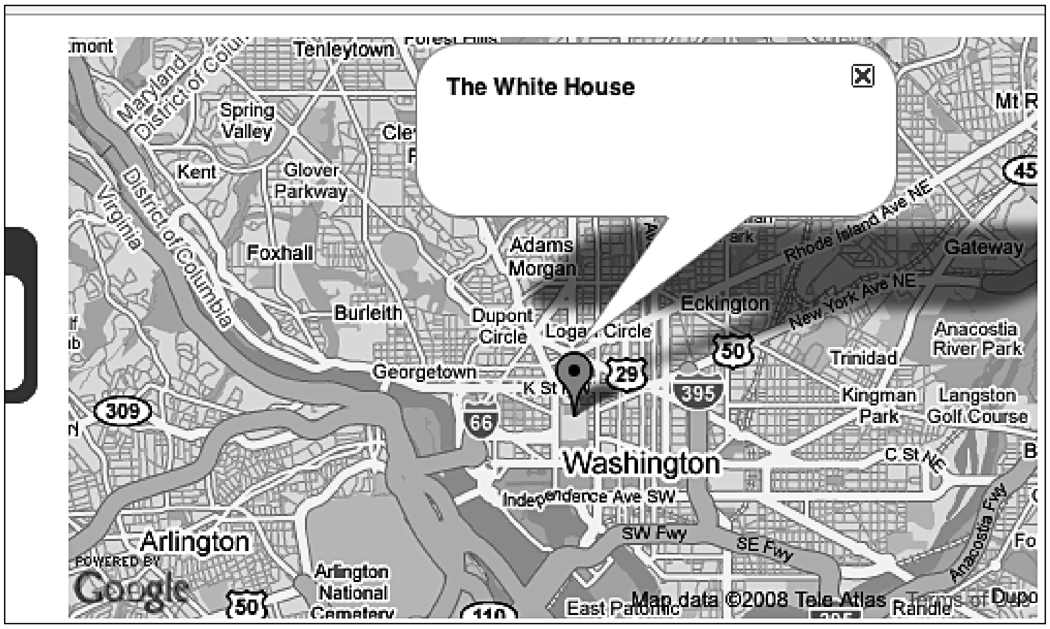
We can make this map slightly more interesting by adding a marker to a specific address. To do so, add the highlighted code below to the `basicmap.js` file:

```
window.addEvent('domready', function() {
    if (GBrowserIsCompatible()){
        var map = new GMap2($('map_canvas'));
        map.setCenter(new GLatLng(38.89, -77.04), 12);

        var whitehouse = new GClientGeocoder();
        whitehouse.getLatLng('1600 Pennsylvania Ave NW',
         function(latlng){
            marker = new GMarker( latlng );
            marker.bindInfoWindowHtml('<strong>The White
             House</strong>');
```

```
                map.addOverlay(marker);
            });

            window.onunload=function(){
            GUnload();
            };

        }
    });
```

This code sets `whitehouse` as an instance of the `GClientGeocoder` class. Next, the `getLatLng()` member function of `GClientGeocoder` is called. The first parameter is the street address to be looked up. The second parameter is an anonymous function where the `GLatLng` object is passed once the address lookup is complete. Within this function, `marker` is set as a new `GMarker` object, which takes the passed-in `latlng` object as a parameter. The `bindInfoWindowHTML()` member function of `GMarker` is called to add an HTML message to appear in a balloon above the marker. Finally, the maker is passed into the `addOverlay()` member function of `GMap2`, to place it on the map.

Save `basicmap.js` and then reload `index.php?option=com_js&task=basicMap`. You should now see the same map, only with a red pin. When you click on the red pin, your map should look like this:



Interactive maps

These two different maps show the basic functionality of getting Google Maps on your own website. These maps are very basic; you could easily create them at `maps.google.com` then embed them in a standard Joomla! article with the HTML code they provide you. However, you would not have the opportunity to add functions that interact with the other elements on your page. To do that, we will create some more HTML code and then write some MooTools-powered JavaScript to bridge our content with Google Maps.

Open the `/components/com_js/js.php` file and add the following `selectMap()` function to the controller:

```
function selectMap()
{
    $key = 'DoNotUseThisKeyGetOneFromCodeDotGoogleDotCom';

    JHTML::_('behavior.mootools');

    $document =& JFactory::getDocument();
    $document->addScript('http://maps.google.com/maps?file=api&v
     =2&key=' . $key);
    $document->addScript(
     JURI::base() . 'components/com_js/selectmap.js');

    ?>
    <div id="map_canvas" style="width: 500px; height: 300px"></div>
    <select id="map_selections">
        <option value="">(select...)</option>
        <option value="1200 K Street NW">Salad Surprises</option>
        <option value="1221 Connecticut Avenue NW">The Daily
         Dish</option>
        <option value="701 H Street NW">Sushi and Sashimi</option>
    </select>
    <?php
}
```

This function is almost identical to `basicMap()` except for two things—`selectmap.js` is being added instead of `basicmap.js`, and a `<select>` element has been added beneath the `<div>`. The `<select>` element has an `id` that will be used in the JavaScript. The options of the `<select>` are restaurants, with different addresses as values. The JavaScript code will bind a function to the `onChange` event so that the marker will move as different restaurants are selected.

To add this JavaScript, create a file named `selectmap.js` in the `/components/com_js` folder, and fill it with the following code:

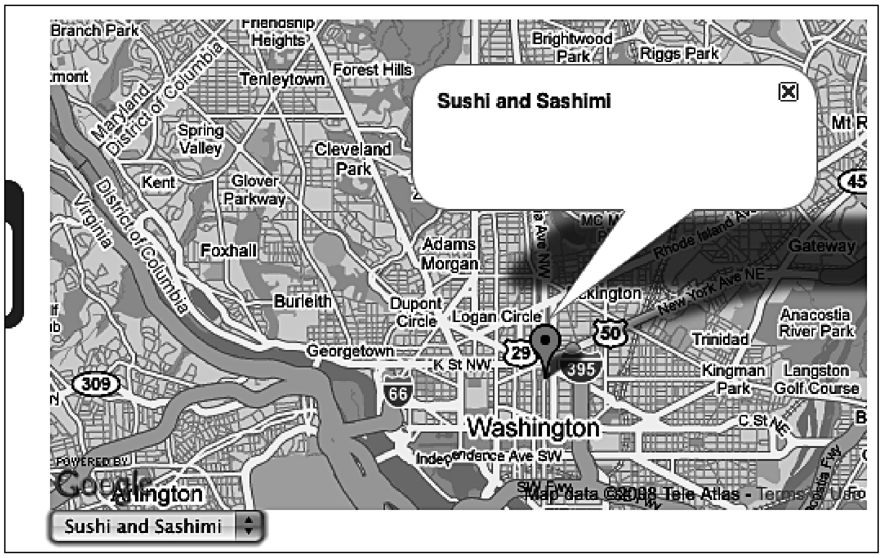
```
window.addEvent('domready', function() {
    if (GBrowserIsCompatible()) {

        var map = new GMap2($('map_canvas'));
        map.setCenter(new GLatLng(38.89, -77.04), 12);

        var restaurant = new GClientGeocoder();

        $('map_selections').addEvent('change', function(){
            if(this.value != '')
            {
                name = this.options[this.selectedIndex].text;
                restaurant.getLatLng(this.value, function(latlng){
                    map_marker = new GMarker( latlng );
                    map.clearOverlays();
                    map.addOverlay(map_marker);
                    map_marker.openInfoWindowHtml('<strong>' + name +
'</strong>');
                });
            }
        });

        window.onunload=function(){
            GUnload();
        };

    }
});
```

This script is similar to the one used for the first two maps, until we get past the call to `map.setCenter()`. The variable `restaurant` is first set as an instance of the `GClientGeocoder()` class. However, member functions are not called on it right away, as it was done in the previous example. This is because we want to wait for the `<select>` element to change before we do anything. The call to `$('map_selections')` finds the `<select>` element, and then `addEvent()` is used to assign a function to the `onChange` event; the string `'change'` is passed as the first parameter.

The function passed in as the second parameter first checks to make sure that the element's value is not null by checking `this.value`. If the value is not null, this function proceeds by getting the option's text and storing it in `name`. Then the `getLatLng()` member function of `GClientGeocoder` is called on `restaurant`. The street address is passed in as the first parameter. Once the latitude and longitude of the address is found, the anonymous function in the second parameter accepts the `GLatLng` object that is returned, using it to set `map_marker` as a new `GMarker` object.

Before adding the marker to the map, the `clearOverlays()` function of the map object is called to remove any previously-placed marker. Then, the marker is passed into the `addOverlay()` function to be placed on the map. Although the marker is now set on the map, we can still interact with it. The `openInfoWindowHtml()` member function of `GMarker` allows us to do this and pass in some HTML with the name of the restaurant.

With the controller function and JavaScript in place and saved, load `index.php?option=com_js&task=selectMap` in your browser. The map should load again, this time with a dropdown box at the bottom. Select one of the restaurants, and then switch to another. If you select **Sushi and Sashimi** from the list, your map should look similar to the following:



These examples scratch the surface of what is possible with the Google Maps API. You can manage layers, add shapes, use custom icons instead of the standard pins, and use many other features to build sophisticated custom maps. For more information, go to `http://code.google.com/apis/maps/documentation/index.html` where you can read more about the available features.

Using jQuery

Another popular JavaScript framework is jQuery. The jQuery framework has functionality that is similar to MooTools, but uses a different style of code. Although they are separate projects, extra care must be taken when using them together. By default, both frameworks automatically reserve `$` to have a special meaning in JavaScript. If you do not take steps to avoid this behavior, one library will overwrite the other's assignments and the scripts will fail.

Fortunately, jQuery has strategies that you can use to avoid this conflict. There are three things that you must do to keep jQuery and MooTools out of each other's way:

- Load jQuery only after MooTools has been loaded
- Call `jQuery.noConflict()` to return control of `$` to MooTools
- Reference jQuery directly in your scripts instead of using `$`

Writing jQuery code

Before avoiding conflicts with MooTools, start by writing some jQuery without MooTools being present. First, go to `jquery.com`, download the latest version of jQuery, and then place it in the `/components/com_js` directory; the production version of the script will be fine. Then, create a file named `jquery-test.js` in the `/components/com_js` folder, and fill this new file with the following code:

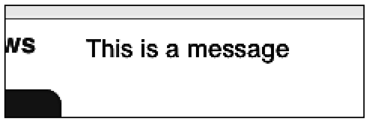
```
$(document).ready(function() {
    $('#message_box').click(function() {
        $(this).addClass('contentheading');
    });
});
```

This is a simple application of jQuery—when the DOM is fully loaded in the browser, the code within the first call to `function()` executes. The code `$('#message_box')` finds the element in the DOM that has an id of `'message_box'`, and uses `click()` to assign an `onClick` JavaScript event to it. Finally, the `addClass()` method is used to add the CSS class `'contentheading'` to the element.

With `jquery-test.js` in place, open the `/components/com_js/js.php` file, and add the following function task to the controller:

```
function useJquery()
{
    $document =& JFactory::getDocument();
    $document->addScript(
     JURI::base() . 'components/com_js/jquery-1.2.6.min.js');
    $document->addScript(
     JURI::base() . 'components/com_js/jquery-test.js');

    ?>
    <p id="message_box">This is a message</p>
    <?php
}
```

First, a reference to the document object is stored in `$document` by using the `getDocument()` member function of `JFactory`. Next, the `addScript()` member function is used to add the jQuery framework and our test script. Finally, a paragraph tag with the id `'message_box'`, and containing a short message, is output. Load `index.php?option=com_js&task=useJquery` in your browser, and click on the **This is a message** link. Your screen should now look similar to this:



Using jQuery with MooTools

The code that we just wrote works fine on its own, but will fail right now if we attempt to also use a MooTools-driven effect. To remedy this, we need to make an adjustment to our script and invoke jQuery's `noConflict` mode. Open the file`/components/com_js/js.php` and add the following function to the controller:

```
    function useJqueryAndMooTools()
    {
        JHTML::_('behavior.tooltip');

        $document =& JFactory::getDocument();
        $document->addScript(
         JURI::base() . 'components/com_js/jquery-1.2.6.min.js');

        $document->addCustomTag('<script type="text/javascript">
         jQuery.noConflict();
         </script>');

        $document->addScript(
         JURI::base() . 'components/com_js/jquery-test.js');

        ?>
        <p id="message_box">This is a message</p>
        <span class="hasTip" title="Click here to go to the home page">
            <a href="index.php">Homepage</a>
        </span>
        <?php
    }
```

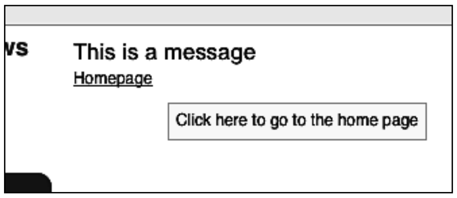
The highlighted portions of `useJqueryAndMooTools()` are the added pieces that differentiate it from `useJquery()`. The call to `JHTML::_('behavior.tooltip');` loads the code necessary for the tooltip, including MooTools. Because MooTools is now being loaded, we use the `addCustomHeadTag()` member function of the document object to make a quick call to `jQuery.noConflict();` so that `$` is left for MooTools. Finally, an anchor tag wrapped in a `<span>` element has been added with the tooltip information.

If you save `js.php` and load `index.php?option=com_js&task=useJqueryAndMooTools` now, you will get an error; we have not yet adjusted `jquery-test.js` to account for the call to `jQuery.noConflict();`. Open `jquery-test.js` and make the highlighted adjustment:

```
jQuery(document).ready(function($) {
    $('#message_box').click(function() {
        $(this).addClass('contentheading');
    });
});
```

Because the `$` shortcut is no longer available, we must call the jQuery function by name. However, we can regain control of the `$` shortcut within our jQuery code. This is because the call to `function()` allows us to pass a reference to the jQuery function into the local scope, with any desired name. Because `$` has been specified as the parameter of `function()`, the rest of the code can use it without further modification.

Load `index.php?option=com_js&task=useJqueryAndMooTools` in your browser, click on the **This is a message** link, then move and the mouse over the **Homepage** link. Your screen should look similar to this:



Always load MooTools first

The trick to using jQuery and MooTools together in Joomla! is to make sure that MooTools loads before jQuery does. In the `useJqueryAndMooTools()` function, if you move the call to `JHTML::_('behavior.tooltip');` any time after the call to `addScript()` that loads in jQuery, an error will occur. Instead, MooTools must load first and define `$`. Then `jQuery.noConflict();` can be called to return `$` back to its original assignment.

This is easily managed when you know and can control every instance where MooTools is used. However, if you are building a self-contained extension for others to use, it is quite possible that they are using other extensions that load MooTools. For instance, if someone has a module for a photo gallery that uses MooTools, the MooTools framework will be added to the `<head>` section of the HTML document after your component with jQuery loads. This will cause a JavaScript error.

The safest way of avoiding this situation is to force the MooTools library to load in your extension before you load jQuery. To do this, simply add `JHTML::_('behavior.mootools');` in your extension before adding the jQuery framework to `<head>`. The call to `JHTML::_('behavior.mootools');` detects whether MooTools has been loaded yet, and loads it now if it has not.

A downside to this workaround is that the MooTools framework will be included even if it is never used. This will slow down the load time of your site slightly and use resources in the JavaScript environment. If you are building an extension that will not be distributed or reused, you can hard-code the references to jQuery in your template after `<jdoc:include type="head" />`. In this scenario, jQuery will be loaded on every page and will eventually be used where you determine. If MooTools is loaded by another extension, it will happen before jQuery is initialized.

Summary

Joomla! is ready for your modern JavaScript needs. By using the MooTools JavaScript framework, you can find elements and assign event handlers to them. You can also use pre-built functions within Joomla! to generate the JavaScript necessary for many typical effects. Regardless of the number of effects used, Joomla! will make sure that the MooTools framework is only loaded once, as long as you use the `JHTML` member functions. You can also use jQuery with MooTools in Joomla!, provided that you take precautions to make sure that the frameworks do not conflict.