3
Using the Tabs Widget

Now that we've been formally introduced to the jQuery UI library, the CSS framework, and some of the utilities, we can move on to begin looking at the individual components included in the library. Over the next six chapters, we'll be looking at the widgets. These are a set of visually engaging, highly configurable user interface widgets.

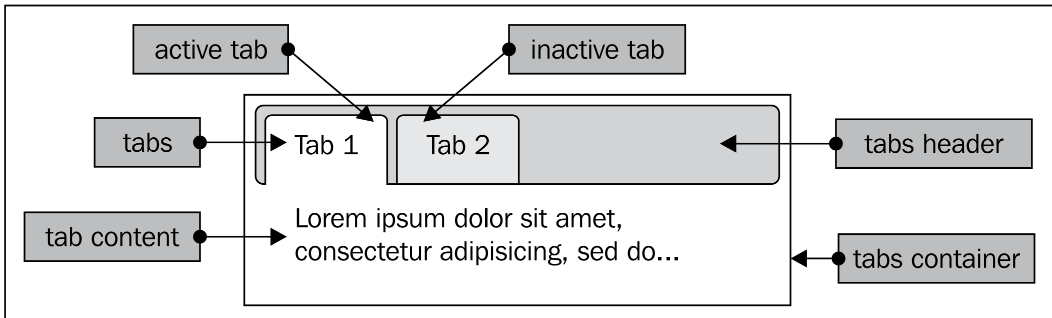
The UI tabs widget is used to toggle visibility across a set of different elements, with each element containing content that can be accessed by clicking on its tab heading. Each **panel** of content has its own tab. The tab headings are usually displayed across the top of the widget, although it is trivial to reposition them so that they appear along the bottom of the widget instead.

The tabs are structured so that they line up next to each other horizontally, whereas the content sections are all set to `display: none` except for the active panel. Clicking a tab will highlight the tab and show its associated content panel, while ensuring all of the other content panels are hidden. Only one content panel can be open at a time. The tabs can be configured so that no content panels are open.

In this chapter, we will look at the following topics:

- The default implementation of the widget
- How the CSS framework targets tab widgets
- How to apply custom styles to a set of tabs
- Configuring tabs using their options
- Built-in transition effects for content panel changes
- Controlling tabs using their methods
- Custom events defined by tabs
- AJAX tabs

The following screenshot shows the different elements that a set of jQuery UI tabs consists of:



A basic tab implementation

The structure of the underlying HTML elements on which tabs are based is fairly rigid, and widgets require a certain number of elements for them to work.

The tabs must be created from a list element (ordered or unordered) and each list item must contain an `<a>` element. Each link will need to have a corresponding element with a specified `id` that is associated with the `href` attribute of the `link`. We'll clarify the exact structure of these elements after the first example.

In a new file in your text editor, create the following page:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Tabs</title>
    <link rel="stylesheet"
      href="css/smoothness/jquery-ui-1.8.9.custom.css">
  </head>
  <body>
    <div id="myTabs">
      <ul>
        <li><a href="#a">Tab 1</a></li>
        <li><a href="#b">Tab 2</a></li>
      </ul>
      <div id="a">This is the content panel linked to the first tab,
        it is shown by default.
      </div>
      <div id="b">This content is linked to the second tab and will
        be shown when its tab is clicked.
```

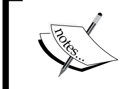
```
    </div>
  </div>
  <script src="development-bundle/jquery-x.x.x,js">
  </script>
  <script src="development-bundle/ui/jquery.ui.core.js">
  </script>
  <script
    src="development-bundle/ui/jquery.ui.widget.js">
  </script>
  <script src="development-bundle/ui/jquery.ui.tabs.js">
  </script>
  <script>
    (function($){
      $("#myTabs").tabs();
    })(jQuery);
  </script>
</body>
</html>
```

Save the code as `tabs1.html` in your `jqueryui` working folder. Let's review what was used. The following script and CSS resources are needed for the default tab widget configuration:

```
jquery.ui.all.css
jquery-x.x.x.js
jquery.ui.core.js
jquery.ui.widget.js
jquery.ui.tabs.js
```

A tab widget is usually constructed from several standard HTML elements arranged in a specific manner:

- An outer container element, which the tabs method is called on
- A list element (`<ul>` or `<ol>`)
- An `<a>` element within an `<li>` element for each tab
- An element for the content panel of each tab



> These elements can be either hardcoded into the page, added dynamically, or can be a mixture of both, depending upon the requirements.

The list and anchor elements within the outer container make the clickable tab headings, which are used to show the content section that is associated with the tab. The `href` attribute of the link should be set to a fragment identifier, prefixed with `#`. It should match the `id` attribute of the element that forms the content section with which it is associated.

The content sections of each tab are created using `<div>` elements. The `id` attribute is required and will be targeted by its corresponding `<a>` element. We've used `<div>` elements in this example as the content panels for each tab, but other elements can also be used as long as the relevant configuration is provided and the resulting HTML is valid. The `panelTemplate` and `tabTemplate` configuration options can be used to change the elements used to build the widget (see the *Configuration* section, later in the chapter for more information).

We link to several `<script>` resources from the library at the bottom of the `<body>` before its closing tag. Loading scripts last, after style sheets and page elements, is a proven technique for improving the apparent loading time of a page, and should therefore be used whenever possible.

After linking first to jQuery, we link to the `jquery.ui.core.js` file that is required by all components (except the effects, which have their own core file), and the `jquery.ui.widget.js` file. We then link to the component's source file, which in this case is `jquery.ui.tabs.js`.

After the three required script files from the library, we can turn to our custom `<script>` element in which we add the code that creates the tabs. We use an anonymous function which is executed immediately (thanks to the extra set of parentheses following the function) and aliases the `$` character.

Aliasing the `$` character in this way is considered a current best-practice, and although not strictly necessary unless writing our own custom jQuery UI plugins, it does make our code more portable by ensuring that other JavaScript libraries, which may use the `$` character, do not break it.

Within this anonymous function we simply call the `tabs()` widget method on the jQuery object, representing our tabs container element (the `<ul>` with an `id` of `myTabs`). When we run this file in a browser, we should see the tabs as they appeared in the first screenshot of this chapter (without the annotations of course).

Tab CSS framework classes

Using Firebug for Firefox (or another generic DOM explorer), we can see that a variety of class names are added to the different underlying HTML elements that the tabs widget is created from.

Let's review these classnames briefly and see how they contribute to the overall appearance of the widget. To the outer container `<div>`, the following classnames are added:

Classname	Purpose
`ui-tabs`	Allows tab-specific structural CSS to be applied.
`ui-widget`	Sets generic font styles that are inherited by nested elements.
`ui-widget-content`	Provides theme-specific styles.
`ui-corner-all`	Applies rounded corners to the container.

The first element within the container is the `<ul>` element. This element receives the following classnames:

Classname	Purpose
`ui-tabs-nav`	Allows tab-specific structural CSS to be applied.
`ui-helper-reset`	Neutralizes browser-specific styles applied to `<ul>` elements.
`ui-helper-clearfix`	Applies the clear-fix as this element has children that are floated.
`ui-widget-header`	Provides theme-specific styles.
`ui-corner-all`	Applies rounded corners.

The individual `<li>` elements that form a part of the tab headings are given the following classnames:

Classname	Purpose
`ui-state-default`	Applies the standard, non-active, non-selected, non-hovered state to the tab headings.
`ui-corner-top`	Applies rounded corners to the top edges of the elements.
`ui-tabs-selected`	This is only applied to the active tab. On page-load of the default implementation, this will be the first tab. Selecting another tab will remove this class from the currently selected tab and apply it to the newly selected tab.
`ui-state-active`	Applies theme-specific styles to the currently selected tab. This class name will be added to the tab that is currently selected, just like the previous classname. The reason there are two class names is that `ui-tabs-selected` provides the functional CSS, while `ui-state-active` provides the visual, decorative styles.

The `<a>` elements within each `<li>` are not given any classnames, but they still have both structural and theme-specific styles applied to them by the framework.

Finally, the panel elements that hold each tab's content are given the following classnames:

Classname	Purpose
`ui-tabs-panel`	Applies structural CSS to the content panels.
`ui-widget-content`	Applies theme-specific styles.
`ui-corner-bottom`	Applies rounded corners to the bottom edges of the content panels.

All of these classes are added to the underlying HTML elements automatically by the library. We don't need to manually add them when coding the page or adding the base markup.

Applying a custom theme to the tabs

In the next example, we can see how to change the tabs' basic appearance. We can override any rules used purely for display purposes with our own style rules for quick and easy customization, without changing the rules related to the tab functionality or structure.

In a new file in your text editor, create the following very small style sheet:

```
#myTabs {
  width:400px; padding:5px; border:1px solid #636363;
  background:#c2c2c2 none;
}
.ui-widget-header {
  border:0; background:#c2c2c2 none; font-family:Georgia;
}
#myTabs .ui-widget-content {
  border:1px solid #aaa; background:#fff none; font-size:80%;
}
.ui-state-default, .ui-widget-content .ui-state-default {
  border:1px solid #636363; background:#a2a2a2 none;
}
.ui-state-active, .ui-widget-content .ui-state-active {
  border:1px solid #aaa; background:#fff none;
}
```

This is all we need. Save the file as `tabsTheme.css` in your `css` folder. If you compare the classnames with the tables on the previous pages, you'll see that we're overriding the theme-specific styles. Because we're overriding the theme file, we need to meet or exceed the specificity of the selectors in `theme.css`. This is why we target multiple selectors sometimes.

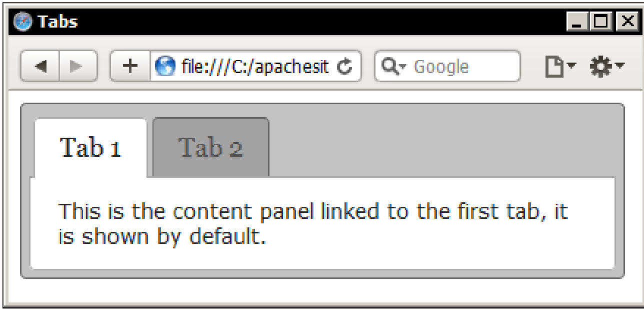
In this example, we override some of the rules in `jquery.ui.tabs.css`. We need to use the `ID` selector of our container element along with the selector from `jquery.ui.theme.css` (`.ui-widget-content`), in order to beat the double class selector `.ui-tabs .ui-tabs-panel`.

Add the following reference to this new style sheet in the `<head>` of `tabs1.html` and resave the file as `tabs2.html`:

```
<link rel="stylesheet" href="css/tabsTheme.css">
```

Make sure the custom style sheet we just created appears after the `jquery.ui.tabs.css` file, because the rules that we are trying to override will be not overridden by our custom theme file if the style sheets are not linked in the correct order.

If we view the new page in a browser, it should appear as in the following screenshot:



Our new theme isn't dramatically different from the default smoothness (as shown in the first screenshot), but we can see how easy it is, and how little code it requires to change the appearance of the widget to suit its environment.

Configurable options

Each of the different components in the library has a series of options that control which features of the widget are enabled by default. An object literal, or an object reference, can be passed in to the `tabs()` widget method to configure these options.

The available options to configure non-default behaviors are shown in the following table:

Option	Default value	Used to...
`ajaxOptions`	`null`	Specify additional AJAX options. We can use any of the options exposed by jQuery's `$.ajax()` method such as `data`, `type`, `url`, and so on.
`cache`	`false`	Control whether the remote data is loaded only once when the page initializes, or is reloaded every time the corresponding tab is clicked.
`collapsible`	`false`	Allow an active tab to be unselected if it is clicked, so that all of the content panels are hidden and only the tab headings are visible.
`cookie`	`null`	Show the active tab using cookie data on page load. The cookie plugin is required for this option to be used.
`disabled`	`false`	Disable the widget on page load. We can also pass an array of tab indices (zero-based) in order to disable specific tabs.
`event`	`"click"`	Specify the event that triggers the display of content panels.
`fx`	`null`	Specify an animation effect when changing tabs. Supply an object literal or an array of animation effects.
`idPrefix`	`"ui-tabs-"`	Generate a unique `ID` and fragment identifier when a remote tab heading's `<a>` element has no `title` attribute.
`panelTemplate`	`"<div></div>"`	Specifying the elements used for the content section of a tab panel.
`selected`	`0`	Show a tab panel other than the first one on page load (overrides the cookie property).

Option	Default value	Used to...
`spinner`	`"Loading…"`	Specify the loading spinner for remote tabs.
`tabTemplate`	`<li><a href="#{href}"> <span>#{label} </span> </a> </li>`	Specify the elements used when creating new tabs. The `#{href}` and `#{label}` strings are replaced by the widget internally when the new tab is created.

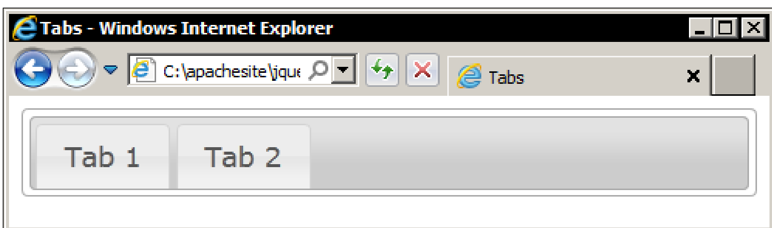
Selecting a tab

Let's look at how these configurable properties can be used. For example, let's configure the widget so that the second tab is displayed when the page loads. Remove the `<link>` for `tabsTheme.css` in the `<head>` of `tabs2.html` and change the final `<script>` element so that it appears as follows:

```
<script>
  (function($){
    var tabOpts = {
      selected: 1
    };
    $("#myTabs").tabs(tabOpts);
  })(jQuery);
</script>
```

Save this as `tabs3.html`. The different tabs and their associated content panels are represented by a numerical index starting at zero. Specifying a different tab to open by default is as easy as supplying its index number as the value for the `selected` property. When the page loads now, the second tab should be selected by default.

Along with changing which tab is selected, we can also specify that no tabs should be initially selected by supplying `null` as the value for this property. This will cause the widget to appear as follows on page load:



Disabling a tab

You may want a particular tab to be disabled until a certain condition is met. This is easily achieved by manipulating the `disabled` property of the tabs. Change the `tabOpts` configuration object in `tabs3.html` to this:

```
var tabOpts = {
  disabled: [1]
};
```

Save this as `tabs4.html` in your `jqueryui` folder. In this example, we remove the `selected` property and add the index of the second tab to the `disabled` array. We could add the indices of other tabs to this array as well, separated by a comma, to disable multiple tabs by default.

When the page is loaded in a browser, the second tab has the classname `ui-widget-disabled` applied to it, and will pick up the disabled styles from `ui.theme.css`. It will not respond to mouse interactions in any way, as shown in the following screenshot:

Transition effects

We can easily add attractive transition effects using the `fx` property. These are displayed when tabs are opened or closed. This option is configured using another object literal (or an array) inside our configuration object, which enables one or more effects. We can enable fading effects, for example, using the following configuration object:

```
var tabOpts = {
  fx: {
    opacity: "toggle",
    duration: "slow"
  }
};
```

Save this file as `tabs5.html` in your `jqueryui` folder. The `fx` object that we created has two properties. The first property is the animation to use when changing tabs. To use fading animations we specify `opacity`, as this is what is adjusted. Toggling the `opacity` simply reverses its current setting. If it is currently visible, it is made invisible and vice-versa.

The second property, `duration`, specifies the speed at which the animation occurs. The values for this property are `slow` or `fast`, which correspond to `200` and `600` milliseconds respectively. Any other string will result in the default duration of `400` milliseconds. We can also supply an integer representing the number of milliseconds the animation should run for.

When we run the file we can see that the tab content slowly fades-out as a tab closes and fades-in when a new tab opens. Both animations occur during a single tab interaction. To only show the animation once, when a tab closes, for example, we would need to nest the `fx` object within an array. Change the configuration object in `tabs5.html` so that it appears as follows:

```
var tabOpts = {
  fx: [{
    opacity: "toggle",
    duration: "slow"
  },
  null]
};
```

The closing effect of the currently open content panel is contained within an object in the first item of the array, and the opening animation of the new tab is the second. By specifying `null` as the second item in the array, we disable the opening animations when a new tab is selected.

We can also specify different animations and speeds for opening and closing animations, by adding another object as the second array item instead of `null`. Save this as `tabs6.html` and view the results in a browser.

Collapsible tabs

By default when the currently active tab is clicked, nothing happens. But we can change this so that the currently open content panel closes when its tab heading is selected. Change the configuration object in `tabs6.html` so that it appears as follows:

```
var tabOpts = {
  collapsible: true
};
```

Save this version as `tabs7.html`. This option allows all of the content panels to be closed, much like when we supplied null to the `selected` property earlier on. Clicking a deactivated tab will select the tab and show its associated content panel. Clicking the same tab again will close it, shrinking the widget down so that only tab headings are visible.

Tab events

The tab widget defines a series of useful options that allow you to add callback functions to perform different actions, when certain events exposed by the widget are detected. The following table lists the configuration options that are able to accept executable functions on an event:

Event	Fired when...
`add`	A new tab is added.
`disable`	A tab is disabled.
`enable`	A tab is enabled.
`load`	A tab's remote data has loaded.
`remove`	A tab is removed.
`select`	A tab is selected.
`show`	A tab is shown.

Each component of the library has callback options (such as those in the previous table), which are tuned to look for key moments in any visitor interactions. Any functions we use within these callbacks are usually executed before the change happens. Therefore, you can return `false` from your callback and prevent the action from occurring.

In our next example, we will look at how easy it is to react to a particular tab being selected, using the standard non-bind technique. Change the final `<script>` element in `tabs7.html` so that it appears as follows:

```
<script>
  (function($){
    var handleSelect = function(e, tab) {
      $("<p></p>", {
        text: "Tab at index " + tab.index + " selected",
        "class": "status-message ui-corner-all"
      }).appendTo(".ui-tabs-nav", "#myTabs").fadeOut(5000, function()
      {
        $(this).remove();
      });
```

```
      },
      tabOpts = {
        select: handleSelect
      }
      $("#myTabs").tabs(tabOpts);
    })(jQuery);
  </script>
```

Save this file as `tabs8.html`. We also need a little CSS to complete this example. In the `<head>` of the page we just created, add the following `<link>` element:

```
<link rel="stylesheet" href="css/tabSelect.css">
```

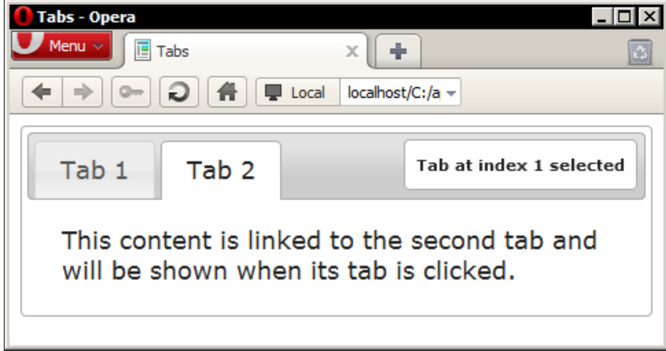
Then in a new page in your text editor, add the following code:

```
.status-message {
  padding:11px 8px 10px; margin:0; border:1px solid #aaa;
  position:absolute; right:10px; top:9px; font-size:11px;
  background-color:#fff;
}
```

Save this file as `tabSelect.css` in the `css` folder. We made use of the `select` callback in this example, although the principle is the same for any of the other custom events fired by tabs. The name of our callback function is provided as the value of the `select` property in our configuration object.

Two arguments will be passed automatically by the widget to the callback function we define, when it is executed. These are the original event object and custom object containing useful properties from the tab, which was selected.

To find out which of the tabs was clicked, we can look at the `index` property of the second object (remember these are zero-based indices). This is added, along with a little explanatory text, to a paragraph element that we create on the fly and append to the widget header:



Whenever a tab is selected, the paragraph before it fades away. Note that the event is fired before the change occurs.

Binding to events

Using the event callbacks exposed by each component is the standard way of handling interactions. However, in addition to the callbacks listed in the previous table, we can also hook into another set of events fired by each component at different times.

We can use the standard jQuery `bind()` method to bind an event handler to a custom event, fired by the tabs widget in the same way that we could bind to a standard DOM event, such as a click.

The following table lists the tab widget's custom binding events and their triggers:

Event	Fired when...
`tabsselect`	A tab is selected.
`tabsload`	A remote tab has loaded.
`tabsshow`	A tab is shown.
`tabsadd`	A tab has been added to the interface.
`tabsremove`	A tab has been removed from the interface.
`tabsdisable`	A tab has been disabled.
`tabsenable`	A tab has been enabled.

The first three events are fired in succession, in the order of events in which they appear in the table. If no tabs are remote, then `tabsselect` and `tabsshow` are fired in that order. These events are sometimes fired before and sometimes after the action has occurred, depending on which event is used.

Let's see this type of event usage in action; change the final `<script>` element in `tabs8.html` to the following:

```
<script>
  (function($) {
    $("#myTabs").tabs();
    $("#myTabs").bind("tabsselect", function(e, tab) {
      alert("The tab at index " + tab.index + " was selected");
    });
  })(jQuery);
</script>
```

Save this change as `tabs9.html`. Binding to the `tabsselect` in this way produces the same result as the previous example, using the `select` callback function. Like last time, the alert should appear before the new tab is activated.

All the events exposed by all the widgets can be used with the `bind()` method, by simply prefixing the name of the widget to the name of the event.

Using tab methods

The tabs widget contains many different methods, which means it has a rich set of behaviors. It also supports the implementation of advanced functionality that allows us to work with it programmatically. Let's take a look at the methods, which are listed in the following table:

Method	**Used to...**
`abort`	Stop any animations or AJAX requests that are currently in progress.
`add`	Add a new tab programmatically, specifying the URL of the tab's content, a label, and optionally its index number as arguments.
`destroy`	Completely remove the tabs widget.
`disable`	Disable a specific tab by passing a zero-based index number to the method, or pass nothing to the method to disable the entire set of tabs.
`enable`	Enable a disable tab by passing a zero-based index number to the method, or enable the entire widget by passing nothing to the method.
`length`	Return the number of tabs in the widget.
`load`	Reload an AJAX tab's content, specifying the index number of the tab.
`option`	Get or set any property after the widget has been initialized.
`remove`	Remove a tab programmatically, specifying the index of the tab to remove. If no index is supplied, the first tab is removed. Tabs can also be removed using their `href` value.
`rotate`	Automatically changes the active tab after a specified number of milliseconds have passed, either once or repeatedly.
`select`	Select a tab programmatically, which has the same effect as when a visitor clicks a tab, based on index number.
`url`	Change the URL of content given to an AJAX tab. The method expects the index number of the tab and the new URL. See also `load` (earlier in this table).
`widget`	Return the element that the `tabs()` widget method is called on.

Enabling and disabling tabs

We can make use of the `enable` or `disable` methods to programmatically enable or disable specific tabs. This will effectively switch on any tabs that were initially disabled or disable those that are currently active.

Let's use the `enable` method to switch on a tab, which we disabled by default in an earlier example. Add the following new `<button>` elements directly after the existing markup for the tabs widget in `tabs4.html`:

```
<button type="button" id="enable">Enable</button>
<button type="button" id="disable">Disable</button>
```

Next, change the final `<script>` element so that it appears as follows:

```
<script>
  (function($){
    $("#myTabs").tabs({
      disabled: [1]
    });
    $("#enable").click(function() {
      $("#myTabs").tabs("enable", 1);
    });
    $("#disable").click(function() {
      $("#myTabs").tabs("disable", 1);
    });
  })(jQuery);
</script>
```

Save the changed file as `tabs10.html`. On the page, we've added two new `<button>` elements—one will be used to enable the disabled tab and the other is used to disable it again.

In the JavaScript, we use the `click` event of the **Enable** button to call the `tabs()` widget method. To do this, we pass the string `enable`, to the `tabs()` method as the first argument. Additionally, we pass the index number of the tab we want to enable as a second argument. All methods in jQuery UI are called in this way. We specify the name of the method we wish to call as the first argument to the widget method.

The `disable` method is used in the same way. Note that a tab cannot be disabled while it is active. Don't forget that we can use both of these methods without additional arguments, in order to enable or disable the entire widget.

Adding and removing tabs

Along with enabling and disabling tabs programmatically, we can also remove them or add completely new tabs on the fly. In `tabs10.html`, remove the existing `<button>` elements and add the following:

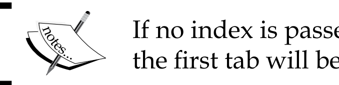
```
<label>Enter a tab to remove:</label>
<input for="indexNum" id="indexNum">
<button type="button" id="remove">Remove!</button><br>
<button type="button" id="add">Add a new tab!</button>
```

Then change the final `<script>` element to this:

```
<script>
  (function($){
    $("#myTabs").tabs();
    $("#remove").click(function() {
      $("#myTabs").tabs("remove", parseInt($("#indexNum").val(),
        10));
    });
    $("#add").click(function() {
      $("#myTabs").tabs("add", "remoteTab.txt", "A New Tab!");
    });
  })(jQuery);
```

Save this as `tabs11.html`. On the page we've added a new instructional `<label>`, an `<input>`, and a `<button>` that are used to specify a tab to remove. We've also added a second `<button>`, which is used to add a new tab.

In the `<script>`, the first of our new functions handle removing a tab, using the `remove` method. This method requires one additional argument—the index number of the tab to be removed. We get the value entered into the textbox and pass it to the method as the argument. The data returned by jQuery's `val()` method is in string format, so we wrap the call in the JavaScript `parseInt` function to convert it.

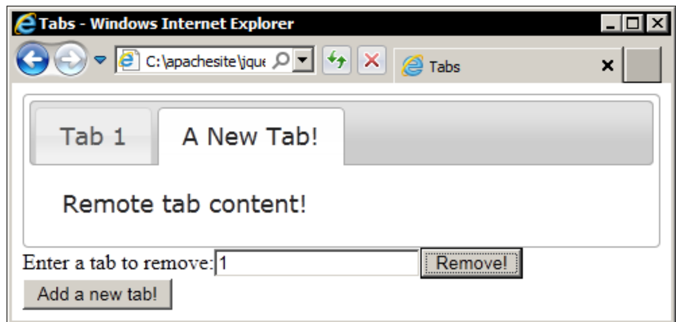


If no index is passed to the `remove` method, the first tab will be removed.

The `add` method that adds a new tab to the widget, can be made to work in several different ways. In this example, we've specified that the content found in the file `remoteTab.txt` should be added as the content of the new tab. In addition to passing the string `add`, and specifying a reference to the file containing the new content, we also specify a label for the new tab in string format as the third argument.

Optionally, we can also specify the index number of where the new tab should be inserted as a fourth argument. If the index is not supplied, the new tab will be added as the last tab.

After adding and perhaps removing some tabs, the page should appear something like this:



> There is a bug in the current version of the tabs widget that causes an extra content panel to be added to the tabs widget, when using the `add` method (see ticket `#5069` at `http://bugs.jqueryui.com/ticket/`). It is this bug that causes the extra space to appear when the new tab is initially added in the previous example.

Simulating clicks

There may be times when you want to programmatically select a particular tab and show its content. This could happen as the result of some other interaction by the visitor.

We can use the `select()` method to do this, which is completely analogous with the action of clicking a tab. Alter the final `<script>` block in `tabs11.html`, so that it appears as follows:

```
<script>
  (function($){
    $("#myTabs").tabs();
    $("#remove").click(function() {
      $("#myTabs")
        .tabs("remove", parseInt($("#indexNum").val(), 10));
    });
    $("#add").click(function() {
     $("#myTabs").tabs("add", "#newTab", "A New Tab!")
        .tabs("select", $("#myTabs").tabs("length") - 1);
    });
  });
</script>
```

Save this as `tabs12.html` in your `jqueryui` folder. Now when the new tab is added, it is automatically selected. The `select()` method requires just one additional argument, which is the index number of the tab to select.

As any tab that we add will, by default (although this can be changed), be the last tab in the interface, and as the tab indices are zero based, all we have to do is use the `length` method to return the number of tabs, and then subtract `1` from this figure to get the index. The result is passed to the `select` method.

Interestingly, selecting the newly-added tab straight away fixes, or at least hides, the extra space issue from the last example.

Creating a tab carousel

One method that creates quite an exciting result is the `rotate` method. This method will make all of the tabs (and their associated content panels) display one after the other automatically.

It's a great visual effect and is useful for ensuring that all, or a lot, of the individual tab's content panels get seen by the visitor. For an example of this kind of effect in action, see the homepage of `http://www.cnet.com`. There is a tabs widget (not a jQuery UI one) that shows blogs, podcasts, and videos.

Like the other methods we've seen, the `rotate` method is easy-to-use. Change the final `<script>` element in `tabs9.html` to this:

```
<script>
  (function($){
    $("#myTabs").tabs().tabs("rotate", 1000, true);
  })(jQuery);
</script>
```

Save this file as `tabs13.html`. We've reverted back to a simplified page with no additional elements other than the underlying structure of the widget. Although we can't call the `rotate` method directly using the initial `tabs()` method, we can chain it to the end like we would with methods from the standard jQuery library.

Chaining UI methods

Chaining widget methods is possible because like the methods found in the underlying jQuery library, they almost always return the jQuery (`$`) object. Note that this is not possible when using getter methods that return data, such as the `length` method.

The `rotate` method is used with two additional parameters. The first parameter is an integer that specifies the number of milliseconds each that tab should display before the next tab is shown. The second parameter is a Boolean that indicates whether the cycle through the tabs should occur once or continuously.

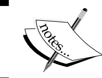
The tab widget also contains a `destroy` method. This is a method common to all the widgets found in jQuery UI. Let's see how it works. In `tabs13.html`, after the widget add a new `<button>` as follows:

```
<button type="button" id="destroy">Destroy the tabs</button>
```

Next, change the final `<script>` element to this:

```
<script>
  (function($){
    $("#myTabs").tabs();
    $("#destroy").click(function() {
      $("#myTabs").tabs("destroy");
    });
  })(jQuery);
</script>
```

Save this file as `tabs14.html`. The `destroy` method that we invoke with a click on the button, completely removes the tab widget, returning the underlying HTML to its original state. After the button has been clicked, you should see a standard HTML list element and the text from each tab, just like in the following screenshot:



Only the original tabs hard-coded in the page will remain if the tabs are destroyed, not those added with the add method.

Getting and setting options

Like the `destroy` method, the `option` method is exposed by all the different components found in the library. This method is used to work with the configurable options and functions in both getter and setter modes. Let's look at a basic example; add the following `<button>` after the tabs widget in `tabs9.html`:

```
<button type="button" id="show">Show Selected!</button>
```

Then change the final <script> element so that it is as follows:

```
<script>
  (function($){
    $("#myTabs").tabs();
    $("#show").click(function() {
      $("<p></p>", {
        text: "Tab at index " + $("#myTabs")
          .tabs("option", "selected") + " is active"
      }).appendTo(".ui-tabs-nav").fadeOut(5000);
    });
  })(jQuery);
</script>
```

Save this file as `tabs15.html`. The `<button>` on the page has been changed, so that it shows the currently active tab. All we do is add the index of the selected tab to a status bar message, as we did in the earlier example. We get the `selected` option by passing the string `selected` as the second argument. Any value of any option can be accessed in this way.

To trigger setter mode instead, we can supply a third argument containing the new value of the option that we'd like to set. Therefore, to change the value of the `selected` option, in order to change the tab being displayed, we could use the following HTML:

```
<label for="newIndex">Enter a tab index to activate</label>
<input id="newIndex" type="text">
<button type="button2" id="set">Change Selected</button>
```

And the following click-handler:

```
<script>
  (function($){
    $("#set").click(function() {
      $("#myTabs")
        .tabs("option", "selected", parseInt($("#newIndex").val()));
    });
  })(jQuery);
```

Save this as `tabs16.html`. The new page contains a `<label>`, an `<input>`, as well as a `<button>` that is used to harvest the index number that the `selected` option should be set to. When the button is clicked, our code will retrieve the value of the `<input>` and use it to change the `selected` index. By supplying the new value we put the method in setter mode.

When we run this page in our browser, we should see that we can switch to the second tab by entering its index number (`1`) and clicking the **Change Selected** button.

AJAX tabs

We saw how we can use the `add` method to add an AJAX tab to the widget dynamically, but we can also add remote content to tabs using the underlying HTML. In this example, we want the tab that will display the remote content to be available all the time, not just after clicking the button. Add the following new `<a>` element to the underlying HTML for the widget in `tabs16.html`:

```
<li><a href="remoteTab.txt">AJAX Tab</a></li>
```

We should also remove the `<button>` from the last example.

The final `<script>` element can be used to just call the `tabs` method; no additional configuration is required:

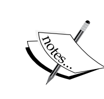
```
$("#myTabs").tabs();
```

Save this as `tabs17.html`. All we're doing is specifying the path to the remote file (the same one we used in the earlier example) using the `href` attribute of an `<a>` element in the underlying markup, from which the tabs are created.

Unlike static tabs, we don't need a corresponding `<div>` element with an `id` that matches the `href` of the link. The additional elements required for the tab content will be generated automatically by the widget.

If you use a DOM explorer, you can see that the file path that we added to link to the remote tab has been removed. Instead, a new fragment identifier has been generated and set as the `href`. The new fragment is also added as the `id` of the new tab (minus the `#` symbol of course) so that the tab heading still shows the tab.

Along with loading data from external files, it can also be loaded from URLs. This is great when retrieving content from a database using query strings or a web service. Methods related to AJAX tabs include the `load` and `url` methods. The `load` method is used to load and reload the contents of an AJAX tab, which could come in handy for refreshing content that changes very frequently.



> There is no inherent cross-domain support built into the AJAX functionality of tabs widget. Therefore, unless additional PHP or some other server-scripting language is employed as a proxy, you may wish to make use of JSON structured data and jQuery's JSONP functionality. Files and URLs should be under the same domain as the page running the widget.

Changing the URL of a remote tab's content

The `url` method is used to change the URL that the AJAX tab retrieves its content from. Let's look at a brief example of these two methods in action. There are also a number of properties related to AJAX functionality.

Add the following new `<select>` element after the tabs widget in `tabs17.html`:

```
<select id="fileChooser">
  <option value="remoteTab1">tabContent.html</option>
  <option value="remoteTab2">tabContent2.html</option>
</select>
```

Then change the final `<script>` element to the following:

```
<script>
  $(function(){
    $("#myTabs").tabs();
    $("#fileChooser").change(function() {
      $("#myTabs").tabs("url", 2, $(this).val());
    });
  });
</script>
```

Save the new file as `tabs18.html`. We've added a simple `<select>` element to the page that lets you choose the content to display in the AJAX tab. In the JavaScript, we set a change handler for the `<select>` and specified an anonymous function to be executed each time the event is detected.

This function simply calls the `url` method, passing in the index of the tab whose URL we want to update and the new URL as arguments.

We'll also need a second local content file. Change the text in the `remoteTab.txt` file and resave it as `remoteTab2.txt`.

Run the new file in a browser and use the dropdown `<select>` to choose the second remote file, then switch to the remote tab. The contents of the second text file should be displayed.

Reloading a remote tab

One thing you may have noticed in the previous example is that if the remote tab was already selected when its URL was changed, the content was not updated until you switched to a different tab and then back to the remote tab. To fix this, we can make use of the `load` method, which reloads a remote tab's content. Update the change event-handler in `tabs18.html` to the following:

```
$("#fileChooser").change(function() {
  $("#myTabs").tabs("url", 2, $(this).val()).tabs("load", 2);
});
```

The `load` method accepts a single argument, which is the index number of the tab whose content we want to reload. We should find now that we can change the URL of the remote tab with the `<select>` element, even while the remote tab is selected and the content will be automatically updated.

The slight flicker in the tab heading is the string value of the `spinner` option that by default is set to `Loading`. We don't really get a chance to see it in full here, as the tab content is changed too quickly as its running locally.

Displaying data obtained via JSONP

Let's pull-in some external content for our final tabs example. If we use the tabs widget, in conjunction with the standard jQuery library `getJSON` method, we can bypass the cross-domain exclusion policy and pull-in a feed from another domain, to display in a tab. In `tabs19.html`, change the tabs widget so that it appears as follows:

```
<div id="myTabs">
  <ul>
    <li><a href="#a"><span>Nebula Information</span></a></li>
    <li><a href="#flickr"><span>Images</span></a></li>
  </ul>
```

```
  <div id="a">
    <p>A nebulae is an interstellar cloud of dust, hydrogen gas, and
      plasma. It is the first stage of a star's cycle. In these re
      gions the formations of gas, dust, and other materials clump
      together to form larger masses, which attract further matter,
      and eventually will become big enough to form stars. The re
      maining materials are then believed to form planets and other
      planetary system objects. Many nebulae form from the gravita
      tional collapse of diffused gas in the interstellar medium or
      ISM. As the material collapses under its own weight, massive
      stars may form in the center, and their ultraviolet radiation
      ionizes the surrounding gas, making it visible at optical wave
      lengths.
    </p>
  </div>
  <div id="flickr"></div>
</div>
```

Next, change the final `<script>` to the following:

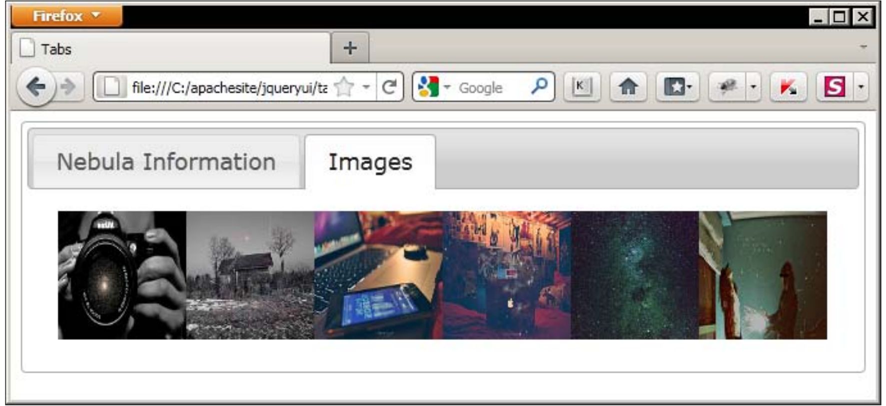
```
<script>
  (function($){
    var img = $("<img/>", {
      height: 100,
      width: 100
    }),
    tabOpts = {
      select: function(event, ui) {
        if (ui.tab.toString().indexOf("flickr") != -1 ) {
          $("#flickr").empty();
          $.getJSON("http://api.flickr.com/services/feeds/
photos_public.gne?tags = nebula&format = json&jsoncallback = ?",
function(data) {
            $.each(data.items, function(i,item){
              img.clone()
              .attr("src", item.media.m).appendTo("#flickr");
              if (i == 5) {
                return false;
              }
            });
          });
        }
      }
    };
    $("#myTabs").tabs(tabOpts);
  })(jQuery);
</script>
```

Save the file as `tabs20.html` in your `jqueryui` folder. We first create a new `<img>` element and store it in a variable. We also create a configuration object and add the `select` event option to it. Every time a tab is selected, the function we set as the value of this option will check to see if it was the tab with an `id` of `flickr` that was selected. If it was, the jQuery `getJSON` method is used to retrieve an image feed from `http://www.flickr.com`.

Once the data is returned, first empty the contents of the **Flickr** tab to prevent a build-up of images, then use jQuery's `each()` utility method to iterate over each object within the returned JSON, and create a clone of our stored image.

Each new copy of the image has its `src` attribute set using the information from the current feed object, and is then added to the empty **Flickr** tab. Once iteration over six of the objects in the feed has occurred, we exit jQuery's `each` method. It's that simple.

When we view the page and select the **Images** tab, after a short delay we should see six new images, as seen in the following screenshot:



Summary

The tabs widget is an excellent way of saving space on your page by organizing related (or even completely unrelated) sections of content that can be shown or hidden, with simple click-input from your visitors. It also lends an air of interactivity to your site that can help improve the overall functionality and appeal of the page on which it is used.

Let's review what was covered in this chapter. We first looked at how, with just a little underlying HTML and a single line of jQuery-flavored JavaScript, we can implement the default tabs widget.

We then saw how easy it is to add our own basic styling for the tabs widget so that its appearance, but not its behavior, is altered. We already know that in addition to this, we can use a predesigned theme or create a completely new theme using ThemeRoller.

We then moved on, to look at the set of configurable options exposed by the tabs' API. With these, we can enable or disable different options that the widget supports, such as whether tabs are selected by clicks or another event, whether certain tabs are disabled when the widget is rendered, and so on.

We took some time to look at how we can use a range of predefined callback options that allow us to execute arbitrary code, when different events are detected. We also saw that the jQuery `bind()` method can listen for the same events if necessary.

Following the configurable options, we covered the range of methods that we can use to programmatically make the tabs perform different actions, such as simulating a click on a tab, enabling or disabling a tab, and adding or removing tabs.

We briefly looked at some of the more advanced functionality supported by the tabs widget such as AJAX tabs and the tab carousel. Both these techniques are easy to use and can add value to any implementation.

In the next chapter, we'll move on to look at the **accordion** widget, which like the tabs widget, is used to group content into related sections that are shown one at a time.