



Prototype & Scriptaculous IN ACTION

SAMPLE CHAPTER

Dave Crane
Bear Bibeault
with Tom Locke

Foreword by Thomas Fuchs



***Prototype and Scriptaculous
in Action***

by Dave Crane
Bear Bibeault
with Tom Locke

Chapter 6

Copyright 2007 Manning Publications

brief contents

PART I GETTING STARTED1

- 1 ■ Introducing Prototype and Scriptaculous 3
- 2 ■ Introducing QuickGallery 26
- 3 ■ Simplifying Ajax with Prototype 45
- 4 ■ Using Prototype's Advanced Ajax Features 71

PART II SCRIPTACULOUS QUICKLY95

- 5 ■ Scriptaculous Effects 97
- 6 ■ Scriptaculous Controls 140
- 7 ■ Scriptaculous Drag and Drop 204

PART III PROTOTYPE IN DEPTH249

- 8 ■ All About Objects 251
- 9 ■ Fun with Functions 277
- 10 ■ Arrays Made Easy 292
- 11 ■ Back to the Browser 325

PART IV ADVANCED TOPICS 357

- 12 ■ Prototype and Scriptaculous in Practice 359
- 13 ■ Prototype, Scriptaculous, and Rails 410

- appendixA* ■ HTTP Primer 443
- appendixB* ■ Measuring HTTP Traffic 458
- appendixC* ■ Installing and Running Tomcat 5.5 469
- appendixD* ■ Installing and Running PHP 477
- appendixE* ■ Porting Server-Side Techniques 489

Scriptaculous Controls



This chapter covers

- Using the Scriptaculous in-place editor controls
- Using the Scriptaculous autocompleter controls
- Using the Scriptaculous slider control

In the previous chapter, we saw how Scriptaculous effects could be used to help reinforce the semantics of user interface interaction, or just to add some dazzle. Scriptaculous also provides extended controls beyond the set that is usually available to web developers. In this and the following chapter, we'll take a look at some of the controls Scriptaculous provides that extend the traditional HTML control set.

Web applications—applications that exist not on the local system but at a remote location and that are accessed through a web browser—have made hefty inroads into spaces where their “desktop” counterparts once ruled supreme. There are many advantages, both to the application service provider and to the client, to delivering applications and services in this manner. But web applications have always suffered from restrictions placed upon them by the limitations of the browsers used to present them. In particular, the set of user interface controls made available to web application authors by the browsers, and as defined by the HTML standards, is very limited compared to the controls generally available to programmers of traditional desktop applications.

However, with advances in browser capabilities, particularly in the areas of JavaScript and CSS, and with the addition of technologies such as Ajax, web application developers have been cleverly bypassing these limitations, using DHTML to create their own controls or to extend the capabilities of the existing controls.

In this chapter, we'll take a look at three control types that Scriptaculous makes available: in-place editors, autocompleters, and sliders.

6.1 Using the sample programs for this chapter

Most of the sample programs used to illustrate the material covered in this chapter require the facilities of back-end resources. For this chapter, these resources take the form of JavaServer Pages (JSP) and Java servlets. As such, you will need to set up an application server capable of serving resources conforming to the Java Servlet 2.4 and JSP 2.0 specifications.

While that may sound daunting if you're not familiar with these types of servers, there's really no need for trepidation. Setting up the freely available Tomcat 5.5 application server is not at all difficult, and details for installing this server, as well as the minimal configuration needed to run the samples in this chapter, are provided in appendix C. If you set up the Tomcat server exactly as outlined in that section, you can view the control panel for this chapter's example at this URL: <http://localhost:8080/sq.chap.6/>. Using `localhost` as the server domain in this URL causes the browser to connect to a server running on your local computer rather than heading out onto the Web to find it. The `:8080` suffix is the port that

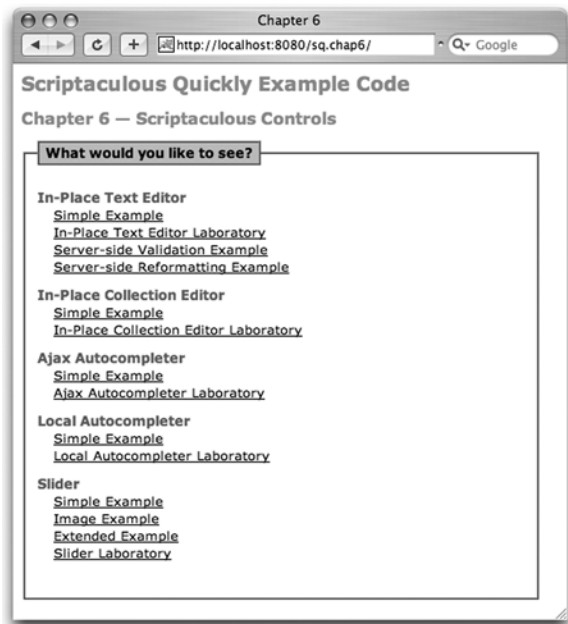


Figure 6.1
The chapter 6 control panel page

Tomcat will be listening on and must be specified (by default, a browser will send HTTP requests to port 80).

As you can see in figure 6.1, this control panel page displays links to the various examples discussed in this chapter. When working through this chapter and its sample programs, you can use this control panel to launch the samples, or you can poke around in the folders directly if you prefer.

Now let's dig in and see what Scriptaculous has to offer for extended controls. We'll start with the in-place text editor.

6.2 *The in-place text editor*

We'll start our journey into extended controls with "in-place editors." Let's look at what is meant by an in-place editor and how it will be useful on our pages.

In many traditional web applications, some read-only data is displayed to the user with an Edit button that redisplay the data for editing (whether in a dialog box, a pop-up window, a floating `<div>` masquerading as a modal dialog box, or even a new page). While this model may be fine in most circumstances, there are times when such heavy-handed interaction is disruptive to the application workflow.

Consider, for example, a situation where a single word needs to be changed (perhaps as part of a spellcheck operation), or a single field needs a badly

formatted value changed. In such circumstances, in-place editing might come in handy. This concept allows an element that normally appears as a simple read-only value to be transformed into a read-write control for editing and to then revert to the read-only format.

Standard HTML provides no such capabilities, but by using DHTML, we can fake it by using the JavaScript and CSS tools at our disposal. Scriptaculous has done a lot of this work for us in the guise of its in-place editor control. Let's see how to use it.

6.2.1 Creating an in-place text editor

Creating a Scriptaculous in-place text editor is relatively simple. We'll follow a paradigm that should be familiar to you by now: an instance of an `Ajax.InPlaceEditor` is created, and some parameters are passed to its constructor, including an anonymous object consisting of a hash (associative array) containing the control's options. A simple example is shown in listing 6.1.

Listing 6.1 Creating an `InPlaceEditor`

```
new Ajax.InPlaceEditor( ❶ The target  
  'theElement',        ← element  
  'transform.jsp',     ❷ The server-  
  {                    ← side resource  
    formId: 'whatever',  
    okText: 'Upper me!',  
    cancelText: 'Never mind' ❸ The options  
  }  
);
```

In this code snippet, an `Ajax.InPlaceEditor` object is created and associated with a target element that is provided as the first of the parameters ❶. In the usual Prototype fashion, this can either be a reference to the element itself or the ID of the target element. As the “Ajax” namespace of the `Ajax.InPlaceEditor` class might lead you to believe, this control connects back to the server using Ajax technology. This connection can be established for two purposes, but the primary purpose is to submit the edited value back to the server for whatever reason the application's author might deem appropriate.

The second parameter to the `Ajax.InPlaceEditor` specifies the URL ❷ to the server-side resource that is contacted when an edited value is completed.

The third parameter specifies a hash of the options to be applied to this instance of the `InPlaceEditor` ❸. We'll be taking a look at the various options in detail shortly.

Now that you know how they're created, let's look at how an `InPlaceEditor` is used in a complete page, as shown in listing 6.2 (`inplace-text/simple-example.html` in the sample code).

Listing 6.2 A simple `InPlaceEditor` example

```
<html>
  <head>
    <title>Simple InPlaceEditor Example</title>
    <script type="text/javascript"
      src="../scripts/prototype.js"></script>
    <script type="text/javascript"
      src="../scripts/scriptaculous.js"></script>
    <script type="text/javascript">
      window.onload = function() {
        new Ajax.InPlaceEditor( ← ① Create the control
          'theElement',
          'transform.jsp',
          {
            formId: 'whatever',
            okText: 'Upper me!',
            cancelText: 'Never mind'
          }
        );
      }
    </script>
  </head>

  <body>
    <div id="theElement">Click me!</div> ← ② Define the element
    </body>
  </html>
```

In this page, we have initialized our simple `InPlaceEditor` in the `onload` function of the page ①, and targeted it to a `<div>` element in the body ②. You can launch this sample from the sample-code control panel for this chapter (see section 6.1). When initially displayed, this rather simple page looks like the one shown in figure 6.2.



Figure 6.2 Simple `InPlaceEditor` before we do anything

When you follow the compelling advice and click the text on the page, the display is much more interesting, as shown in figure 6.3. Note that the `<div>` element has been replaced (at least visually) with a text entry control, a button, and an active link. The text of the button and active link match those that we specified via the options hash.

If we don't type anything new into the text box and click the Upper me! button, we find that the text reverts to a read-only element, as show, in figure 6.4. But the text has been uppercased! How did that happen?

Recall that when we constructed the `InPlaceEditor`, we specified the `transform.jsp` URL as the second parameter to the constructor. This JSP page is activated with the value of the text control when the button is clicked, and its response value is taken as the new value of the control.

The `transform.jsp` source is shown in listing 6.3.

Listing 6.3 Code for the `transform.jsp` page

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions"
    prefix="fn" %>
<% Thread.sleep(3000); %>
${fn:toUpperCase(param.value)}
```

This rather trivial JSP page converts the value of the query parameter with the key "value" to its uppercase equivalent, and writes the result back to the response. The body of the response becomes the new value for the control. (The call to the `sleep()` method creates an artificial 3-second delay to simulate the lag that might occur if this interaction occurred over a network connection rather than on a local server).

You cannot create an `InPlaceEditor` that is disconnected from the server. Specifying a null, empty string, or anything other than a valid server-side resource for the second parameter to the constructor will not create an `InPlaceEditor` that does not contact the server when the button is clicked (or the Enter key is pressed).



Figure 6.3 Simple `InPlaceEditor` in its active editable state



Figure 6.4 Simple `InPlaceEditor` with the changes that were made

If you want an `InPlaceEditor` that has no apparent server-side effects, you can create a simple “reflector” resource that simply returns whatever value is fed to it. In JSP 2.0, something as simple as the following single-line JSP page would do the job:

```
#{param.value}
```

Now that we can create an `InPlaceEditor`, let’s look at how we can control it.

Script control of the `InPlaceEditor`

Once we’ve got an editor on our page, we will want to exert some control over it with our code. If you store a reference to the editor instance, like this,

```
var inPlaceEditor = new Ajax.InPlaceEditor( ...
```

you will be able to use scripts to control the activity of the editor.

For example, if you need to force the editor into active mode, you can do so with the editor’s `enterEditMode()` function, as shown in the following code snippet:

```
function onSomeEvent(event) {  
    //do something  
    inPlaceEditor.enterEditMode();  
}
```

This function takes an optional parameter that accepts the current event and cancels that event. If, for example, the preceding function were triggered by the user clicking a link, passing the click event to the `enterEditMode()` function would cancel loading the link.

You can also disconnect the `InPlaceEditor` functionality from its associated element if, for whatever reason, your page needs to prevent the target element from entering active mode. You would do so by invoking the `dispose()` function on the editor instance as follows:

```
inPlaceEditor.dispose();
```

Now that you’ve seen a simple example, let’s take a look at how we can use the option hash to further tailor the control to our needs.

6.2.2 The `InPlaceEditor` options

There is a long list of options that can be applied to an `InPlaceEditor` in order to assert control over its appearance and operation, as shown in table 6.1. Each of these options will be discussed in more detail in the following subsections.

Table 6.1 The Ajax.InPlaceEditor options

	Option	Description
Appearance options	okButton	A Boolean value indicating whether or not an OK button is to be shown. Defaults to <code>true</code> .
	okText	The text to be placed on the OK button. Defaults to "OK".
	cancelLink	A Boolean value indicating whether a cancel link should be displayed. Defaults to <code>true</code> .
	cancelText	The text of the cancel link. Defaults to "cancel".
	savingText	A text string displayed as the value of the control while the save operation (the request initiated by clicking the OK button) is processing. Defaults to "Saving...".
	clickToEditText	The text string that appears as the control's tooltip upon mouse-over.
	rows	The number of rows that will be displayed when the edit control is active. Any number greater than 1 causes a text area element to be used rather than a text field element. Defaults to 1.
	cols	The number of columns displayed when in active mode. If omitted, no column limit is imposed.
	size	Same as <code>cols</code> but only applies when <code>rows</code> is 1.
	highlightcolor	The color to apply to the background of the text element upon mouse-over. Defaults to a pale yellow.
	highlightendcolor	The color that the highlight color fades to as an effect. Note that some browsers don't entirely support this option.
loadingText	The text to appear within the control during a load operation. (We'll be looking at that option shortly.) The default is "Loading...".	
CSS styling and DOM ID options	savingClassName	The CSS class name applied to the element while the save operation is in progress. This class is applied when the request to the saving URL is made, and it is removed when the response is returned. The default value is "inplaceeditor-saving".
	formClassName	The CSS class name applied to the form created to contain the editor element. Defaults to "inplaceeditor-form".
	formId	The ID applied to the form created to contain the editor element.

Table 6.1 The `Ajax.InPlaceEditor` options (*continued*)

	Option	Description
Callback options	<code>callback</code>	A JavaScript function that is called just prior to submitting the save request in order to obtain the query string to be sent to the request. The default function returns a query string equating the query parameter value to the value in the text control
	<code>onComplete</code>	A JavaScript function that is called upon successful completion of the save request. The default applies a highlight effect to the editor.
	<code>onFailure</code>	A JavaScript function that is called upon the failure of the save request. The default issues an alert showing the failure message.
Other options	<code>loadTextURL</code>	The URL of a server-side resource to be contacted in order to load the initial value of the editor when it enters active mode. By default, no back-end load operation takes place, and the initial value is the text of the target element.
	<code>externalControl</code>	An element that is to serve as an external control that triggers the editor into active mode. This is useful if you want another button or other element to trigger editing the control.
	<code>ajaxOptions</code>	A hash object that will be passed to the underlying Prototype Ajax object to use as its options hash.

While exploring these options in greater depth, it will be handy to have some code that will let us see the options in action. So, with great fanfare, we introduce the `InPlaceEditor` Lab page.

The `InPlaceEditor` Lab

In order to demonstrate the options available for the `InPlaceEditor` control, the `InPlaceEditor` Lab page has been included in the source code for this chapter. You can launch this lab page via the control panel for this chapter's example programs. When you initially display the lab page, it will appear as shown in figure 6.5.

The `InPlaceEditor` Options section allows you to play around with some of the more commonly used options for this control. Once you have entered the options that you'd like, click the `Create Editor` button to create the editor control, which will appear below the `InPlaceEditor` Options section.

The `Applied Options` section will display the options that you have selected, and the `Status` area uses an `onComplete` callback function to display the value of the control element's `innerHTML` property.

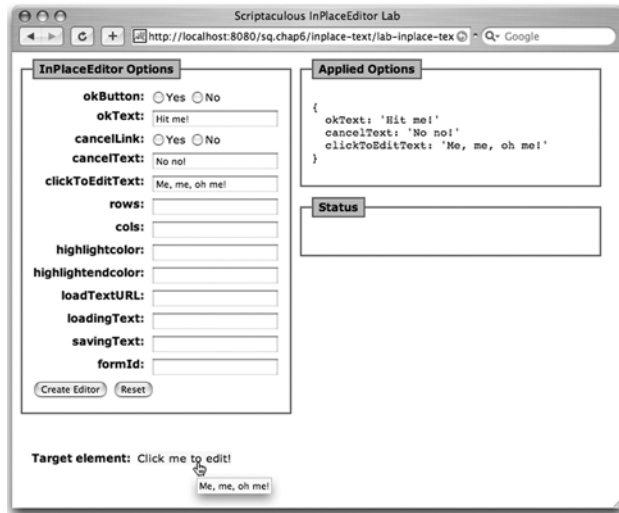


Figure 6.5
The Scriptaculous
InPlaceEditor Lab in action

A typical usage scenario is shown in figure 6.6.

As you read through the following sections on the options, please experiment with them in the lab page until you are familiar with their actions. You are welcome to make a copy of the lab, or of the simple example pages, and edit to your heart's content, such as to add options that are not currently included in the lab page.

Now let's dig in and get to know the various options.

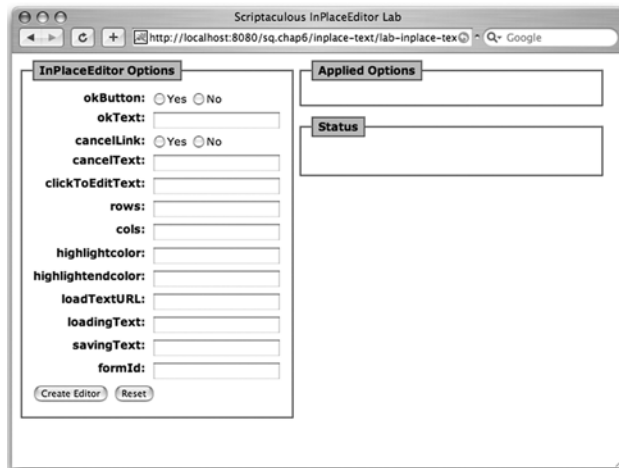


Figure 6.6
Initial state of the
Scriptaculous InPlace-
Editor Lab page

Controlling the appearance of the editor

The set of `InPlaceEditor` Options in the lab allows you to control which elements of the editor are displayed, and what they look like.

The `okButton` option allows you to include or omit the OK button, which saves the editor value to the server-side resource, specified as the editor object's second constructor parameter. When the OK button is omitted, the user can still initiate the save action by pressing the Enter key.

When an OK button is displayed, its text value is controlled by the `okText` option. By default, the OK button is displayed and contains the text "ok". We saw an example of supplying alternate text in listing 6.2.

The `cancelLink` option controls a link that cancels the editor without invoking a save operation. The text of the link, which appears unless explicitly disabled, defaults to "cancel" and can be controlled with the `cancelText` option. Unfortunately, there is no option to make the cancel link a button rather than a text link.

While the editor is invoking the save operation, the text "Saving..." is displayed within the editor control. You can override this value with the `savingText` option.

Normally, when the editor is placed into edit mode, an HTML input element of text type is used as the active control. When an explicit value greater than 1 is supplied for the `rows` options, however, an HTML `<textarea>` element with the specified number of rows is used instead.

Regardless of how many rows are specified (or whether `rows` is left to the default of 1), the `cols` option can be used to control the width of the control. When a single row is being used, the `size` attribute of the text area is set to the value of the `cols` option. If a multirow text area is used, the `cols` attribute of the text area is set to the value of the `cols` option. Note that in neither case does this setting affect the number of characters that can be entered into the control. It merely sets the display width of the element (which arguably would be more appropriately set using CSS rules, as outlined in the next section).

You can affect how the Scriptaculous Highlight effect is applied to the `InPlaceEditor` with the `highlightcolor` and `highlightendcolor` options, although the results seem to vary rather broadly across different browsers. We saw the Scriptaculous Highlight effect in section 5.5.3 of the previous chapter.

The `loadingText` option is similar to the `savingText` option in that it specifies the text to be displayed while a back-end operation is taking place. This text is displayed only if the `loadTextUrl` option is also set, as discussed a little later in this section.

Specifying CSS and DOM ID options

The Scriptaculous `InPlaceEditor` control also gives you a small degree of control over some of the DOM IDs used, as well as CSS-style class names.

Unless you specify otherwise, the name of the form generated to hold the control (when in active mode) is formed by suffixing the text “-inplaceeditor” to the end of the ID of the target element. So if the target element has an ID of `targetElement`, the name assigned to the form would be `targetElement-inplaceeditor`. If that doesn’t suit your fancy for whatever reason, the `formId` option can be used to assign another name to the form.

One reason you might want to replace the default form name is if you are planning to access the form or its elements via JavaScript. The hyphen in the default name makes referencing the form within JavaScript expressions rather problematic. A reference such as this,

```
var form = document.targetElement-inplaceeditor;
```

will result in an error, since the hyphen will be construed as a subtraction operator.

We could solve this problem by using the general referencing notation:

```
var form = document['targetElement-inplaceeditor'];
```

Or we could simply use the `formID` option to give the form a name we like better.

There may be times when you want the element’s appearance to change while the save operation is taking place (above and beyond merely changing the text to “Saving...” or whatever else you specified with the `savingText` option). To make this possible, the `InPlaceEditor`’s code adds a class name to the element while a save operation is under way, and then removes it when the save operation completes. By default, the name of this class is `inplaceeditor-saving`. Should that prove undesirable or unwieldy, or if you already have an existing CSS class name that you’d rather use, you can override this name with the `savingClassName` option.

It is important to note that this class name is added to the list of class names for the element. You can rest assured that any class names that you have assigned to the element for other purposes will not be replaced by the class name that Scriptaculous adds to the element during a save operation.

To help you style the contents of the form created while the control is in active mode, Scriptaculous assigns a class name of `inplaceeditor-form` to the form. The `formClassName` option can be used to specify a different class name for the form.

Now let’s see what our controls are doing, so that we can react appropriately to the various state changes that the controls will go through as they are used on the page.

Using the callback options

When a save operation is initiated (either by clicking the Save button or pressing the Enter key while the editor's text field has focus), a JavaScript function is called to generate the query string that will be applied to the server-side post operation. Unless you specify otherwise with the `callback` option, an internal method that serializes the generated form is called. This method uses the `Form.serialize()` method from the Prototype library.

If you'd like to replace the callback function with one of your own (so that you can perform some custom activity before initiating the save operation) you must similarly return a suitable query string as the result of the callback, in order for the save operation to proceed normally. If you specify a custom callback function, it will be passed a handle to the generated form, so producing an appropriate querystring is rather simple, as shown in listing 6.4.

Listing 6.4 A custom callback function for the `InPlaceEditor`

```
function myVeryOwnCallback( form ) {  
  //  
  // custom actions performed here...  
  //  
  return Form.serialize(form);  
}
```

This custom callback function also permits you to change the name of the query parameter used to pass the entered data to the server-side resource. The `InPlaceEditor` always creates its text input control with a name of `value`, and there is no analogous option to `formId` to reassign this name. But you can use the `callback` option to force the query parameter name to be anything you choose.

Let's say that you'd rather that the query parameter passed to your server-side resource be named `steve` rather than `value`. You could replace the default callback with your own function, as shown in the following fragment from the options hash:

```
callback: function(form) {  
  return 'steve=' + form.value.value;  
}
```

You can also register callback functions to be invoked after the save operation has completed. The `onComplete` option specifies a callback that will be invoked if the save operation returns successfully, and the `onFailure` option registers a callback to be invoked in the event of a failure.

The `onComplete` callback is called with two parameters: a reference to the `XMLHttpRequest` instance used to make the save post, and a reference to the target element. If no value is specified for the `OnComplete` option, the `InPlaceEditor` uses an internal function that applies a `Scriptaculous Highlight` effect to the target element using the value specified in the `highlightcolor` option.

The `onFailure` option is invoked if the response to the save post returns an error code; the default function issues an alert with the failure code provided in the `XMLHttpRequest` instance. This callback function is passed a single parameter consisting of the reference to the `XMLHttpRequest` instance.

You can use either of these callback options to customize what happens upon success or failure. For example, alerts can be a rather annoying means of issuing messages. If we had a `<div>` element on a page designed to contain such messages, let's say with an ID of `messageArea`, we could replace the default `onFailure` callback function with one of our own:

```
onFailure: function(xmlReq) {
    $('messageArea').innerHTML = xmlReq.responseText.stripTags();
}
```

Note the use of the handy `stripTags()` method from the `String` extensions of the `Prototype` library—they will remove any pesky HTML markup in the error message.

Now that we know how to keep track of what's going on with our editors, it's time to see how we can exert further control over them. For example, we can control how they initially get loaded.

Loading text from the server

Unless you specify otherwise by using the `loadTextURL` option, the initial value of the text field when the control becomes active will be the content of the target element.

You can specify the URL of a server resource to provide a text value for the `loadTextURL` option, but this post is not passed any data from the client, such as the content of the target element. How can the server-side resource know what text to load, if it doesn't have any client-side contextual information?

Fortunately, the `ajaxOptions` option comes to our aid here. This option specifies another hash object that is passed to the underlying `Prototype Ajax.Request` object that is used to make the server-side post. By specifying the `parameters` option to this hash, we can cause the value of the text field to be passed to the back-end resource:

```
ajaxOptions: {
  parameters: 'initialValue='+$('targetElement').innerHTML
},
```

This will cause the content of the target element (in this case, assume it's named `targetElement`) to be passed as a query parameter named `initialValue`. Bear in mind that this query parameter will be passed to all Ajax operations, including the save operation. Be sure that the query parameter name you choose for controlling the load operation will not interfere with any required for the save operation.

Note that the `ajaxOptions` option also allows you to pass the value back as any query parameter name you want, without forcing you to override the `callback` option.

Now that you've learned about the options to the `InPlaceEditor`, let's see how it all works in a few common scenarios.

6.2.3 Some usage examples

One of the major features of the Scriptaculous `InPlaceEditor` is its connection to a back-end resource. The examples we'll explore in this section will exploit that capability to perform server-side operations on the data that is entered.

First, we'll couple an `InPlaceEditor` with a server-side resource that will validate the value typed into the control. Then we'll enhance that example by adding a data-reformatting operation, demonstrating the level of server-side control that we can exert upon the control's data.

If you haven't already set up Tomcat 5.5 (or an equivalent application server) as outlined in appendix C, now is the time to catch up.

Server-side setup for the examples

So far in this chapter we have employed only simple JSP files as server-side resources. This was quick and easy—we just dropped the JSP files into the same folder as the page we were testing, and used page-relative referencing to address them. However, using JSP files for anything other than presentation is a rather egregious violation of currently accepted best practices. In a properly structured MVC (Model-View-Controller) web application, JSP pages should be relegated to presentation duty only; servlets are the more appropriate mechanism for any heavy-duty server-side operations. So in this section, we will set up and employ a few servlets to do our back-end operations.

Because servlets aren't simple files like HTML and JSP pages, the type of page-relative addressing we have been using up to this point just won't cut it anymore. We need to employ server-relative addressing which begins with the context path of the web application. (See section C.3 in appendix C for an explanation of the context path concept.) It is rightly considered a poor practice to hard-code context paths into pages. Rather, the context path should be rendered programmatically in

the pages so that the web application will work as expected, regardless of what context path was chosen for the application.

To that end, our examples are JSP pages rather than static HTML pages, as most of our other examples have been. If you are operating in an environment where servlets are being employed as back-end resources, you are likely to be familiar with JSP pages being used for presentation.

Within the JSP pages discussed in this section, only a single JSP-ism is utilized: a JSP Expression Language snippet that automatically resolves to the context path for the application. That snippet looks like this:

```
${pageContext.request.contextPath}
```

The details of how and why this expression resolves to the context path of the web application are beyond the scope of this book, but any modern JSP book (or the JSP specification itself) will explain this concept in great detail.

Because this is the only JSP mechanism employed on the pages, you can turn them into simple HTML pages by replacing the expression (including the `#{` and `}` delimiters) with the hard-coded context path of the application.

The Java code for the servlets themselves can be found in the `WEB-INF/src` folder of the chapter 6 web application. The source code for a web application would not normally be included with the structure of the web application like this—we’ve just included it here so you can review the servlet source code if you wish.

Within the `WEB-INF` folder, you will also find a file named `web.xml`. This is the deployment descriptor for the application, and within it you will find declarations that define the servlets to the Tomcat container, and that specify the URLs used to access them.

Performing server-side validation with an InPlaceEditor

Now that we’re all set up to run our examples, let’s use an `InPlaceEditor` to perform validation. In our first example, we’ll use an `InPlaceEditor` to accept an email address, then send it back to the server for validation.

The JSP file for this example is named `validation-example.jsp`, and you’ll find it with the web application code for this chapter, in the `inplace-editor` folder. Open this page in your browser. Be sure that the page is being served by your local Tomcat instance—just opening the file in the browser by dragging the file into it will not cause the active resources to be interpreted.

The page you will see is quite simple, consisting of a label, “Email address:”, and a `` element, which is the target element for the `InPlaceEditor`. Figure 6.7 shows this page when the editor is active.

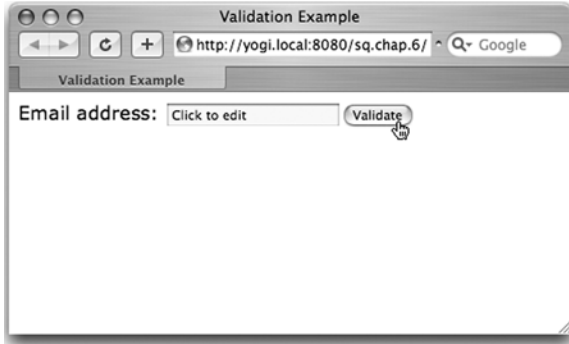


Figure 6.7
The `InPlaceEditor` validation page before any editing

In this example, the text typed into the text box will be validated by a server-side resource. If the entry passes validation, the editor will revert to display mode as expected. However, if the entry fails validation, the server will indicate that, and our code will place a new style class on the text, showing the user that the entry is not valid. Sounds simple enough! The code for the page is shown in listing 6.5.

Listing 6.5 Source for the `InPlaceEditor` validation example

```

<html>
  <head>
    <title>Validation Example</title>
    <script type="text/javascript"
      src="../scripts/prototype.js"></script>
    <script type="text/javascript"
      src="../scripts/scriptaculous.js"></script>
    <script type="text/javascript">
      window.onload = function() {
        new Ajax.InPlaceEditor(
          'emailField',
          '${pageContext.request.contextPath}/checkEmailAddress',
          {
            okText: 'Validate',
            cancelLink: false,
            onComplete: function(req,element) {
              if (req.getResponseHeader('x-valid')=='false') {
                $('emailField').addClassName('fieldInError');
              }
            },
            callback: function(form,entry) {
              $('emailField').removeClassName('fieldInError');
              return 'value=' + entry;
            }
          }
        );
      };
    
```

1 Create `InPlaceEditor`
 2 Specify the server-side resource
 3 Define completion handler
 4 Define callback handler

```

    }
  </script>
  <style type="text/css">
    .fieldInError {
      border: 1px solid maroon;
      color: maroon;
    }
    .inplaceeditor-form {
      display: inline;
    }
  </style>
</head>

<body>

  <p>
    <label>Email address:</label>
    <span id="emailField">Click to edit</span>
  </p>

</body>

</html>

```

5 Apply error styling

6 Apply inline styling to the form

Declare target element

By now, most of the code on this page should be at least somewhat familiar. The page imports the Prototype and Scriptaculous libraries as usual, and, as in prior examples, we set up the InPlaceEditor within the `onload` handler for the page. The creation of the editor itself ❶ specifies the `` element with the ID of `emailField` as its target element.

The parameter for the server-side resource ❷ specifies this odd-looking URL:

```
'${pageContext.request.contextPath}/checkEmailAddress',
```

The construct within the `${}` delimiters was explained in section 6.2.3 and will resolve to the context path for the application (to `/sq.chap.6` if you set up the web application exactly as recommended).

The remaining portion of the URL, `/checkEmailAddress`, is the servlet path. This path was set up in the `web.xml` deployment descriptor as the mapping for the servlet named `CheckEmailAddressServlet`. Note that this mapping does not correspond to physical file on the file system (as would be expected with HTML and JSP files), but to the named servlet class on the Java classpath. (In this class, the class file can be found in the folder hierarchy rooted at `WEB-INF/classes`). We'll take a brief look at that servlet later in this section.

One of the more interesting aspects of this example is the `onComplete` callback specified for the editor ❸. Remember that this function is called upon successful completion of the save operation performed by the servlet. The `XMLHttpRequest` instance and target element are passed to the callback function as parameters, and the first thing our handler does is check the `XMLHttpRequest` object for a special response header named “x-valid”. This header indicates whether or not the text passed back to the servlet has passed validation.

Wait a minute! If the validation fails, shouldn’t the servlet simply return an error response that would trigger an `onFailure` handler? Well yes, it could have been written that way. But there are a number of issues that can be avoided by not doing so. By not returning an error code, it’s easy for our code to distinguish between actual server errors (servlet not found, forbidden access, and the like) and validation failures. Even though we are calling them validation failures, they are not really errors in the request/response cycle—they are one of the expected, successful results, and as such shouldn’t use a response error code.

We also don’t want to use the body of the response to somehow indicate a validation failure. The body of the response is the value that will be placed into the editor control, and we don’t want to mess around with that. If the user typed in a long string that contained one wrong character, he or she would be mighty annoyed if we didn’t reload that string back into the editor, requiring it to be completely retyped.

This is why we set up a response header to return the validation status—we didn’t want to use a response error code, and we didn’t want to use the response body. We named the header “x-valid”, using the prefix “x-” to avoid accidentally stepping on a response header name defined by the HTTP protocol (none of which begin with “x-”), and we can use the `getResponseHeader()` method of `XMLHttpRequest` to get its value.

If the value is the string “false”, as set by the servlet in the case of validation failure, the Prototype `addClassName()` method is called to add the CSS `fieldInError` class name to the target element. This CSS class ❹ places a maroon border around the element and changes the color of the text to indicate that the entry is in error. More elaborate error indications are possible, of course (such as adding flyouts that describe the error, issuing alerts, adding error icons, and so on).

After the user makes a correction to the entry and resubmits the value, the default callback handler is replaced with one of our own ❺ that removes the `fieldInError` class just before the save operation is carried out.

One last point before we take a quick gander at the servlet: `<form>` elements are by default block elements. Left to its own devices, our `InPlaceEditor` would jump to

the next “line” under the label when placed into active mode due to the dynamic creation of the form that encloses the text field. To prevent that, we specify a CSS rule that changes the form from a block element to an inline element **6**. This keeps the InPlaceEditor, well, in place when it makes the transition to active mode.

On the server side, a servlet that performs the validation makes use of Java regular-expression processing to validate an email address. Since the subject of regular expressions is clearly beyond the scope of this discussion, the servlet uses a very simple expression that would never be used in an actual production system.

The servlet code is shown in listing 6.6.

Listing 6.6 The CheckEmailAddressServlet source code

```
package org.bibeault.sq.chap6;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.regex.Pattern;
import java.io.IOException;

public class CheckEmailAddressServlet extends HttpServlet {

    private static final String KEY_VALUE = "value";
    private static final String REGEX_EMAIL =
        "(\\w)+@(\\w)+\\. (\\w)+";
    private static final String HEADER_VALID = "x-valid";

    public void doGet( HttpServletRequest request,
                      HttpServletResponse response )
        throws IOException {
        String value = request.getParameter( KEY_VALUE );
        boolean ok = Pattern.compile( REGEX_EMAIL )
            .matcher( value ).matches();
        response.setHeader( HEADER_VALID, String.valueOf( ok ) );
        response.getWriter().write( value );
    }

    public void doPost( HttpServletRequest request,
                       HttpServletResponse response )
        throws IOException {
        doGet( request, response );
    }
}
```

Without belaboring how this servlet functions, we can see that the `doGet()` method obtains the value of the request parameter named `value`, and matches it against a simple (borderline brain-dead) pattern. The “x-valid” response header is set to `true` or `false` depending upon the outcome of the match, and the original value is written as the response body. The `doPost()` method is written in terms of the `doGet()` method so that the servlet will respond to GET and POST requests in the same manner.

While this is a simple example, there are two important ideas that it introduces: a servlet can be used as a server-side resource for an `InPlaceEditor` (or for any Scriptaculous control that initiates server-side requests), and custom response headers can be used to pass ancillary data back to the client without usurping either response error codes or the response body.

This is impressive stuff! We can use the power of the server to perform validations that would be difficult or impossible if we were limited to merely validating on the client. Now let’s take a look at how the server can play a more active role in extending the capabilities of our controls by actively reformatting the data.

Performing server-side reformatting with an `InPlaceEditor`

In the previous example, we used a server-side resource (implemented as a servlet) to perform a validation check. Regardless of whether the user typed in a valid entry, the entry was returned (and restored to the `InPlaceEditor`) exactly as the user entered it. In this example, we’ll enhance our server-side check to go beyond mere validations and reformat valid entries for display.

Rather than an email address, this page will accept a 10-digit North American phone number whose standardized format is (999)555-1212, where 999 is the area code, and the remaining digits are the phone number within that area. It’s mighty unfriendly to force users to type in all that punctuation, so we’ll allow them to enter the digits in any way they want. By ignoring anything but digits in the input, we can allow them to enter 9995551212, 999-555-1212, 999.555.1212, or any other variation, as long as there are exactly 10 digits in the input. Upon receiving a valid entry, the page will redisplay the entry in the standard format, regardless of which format the user entered. That’s nice. And users like nice.

The JSP page for this example is named `reformat-example.jsp`, and it is found in the `inplace-editor` folder of the web application. This page is almost identical to one in the validation example, except for simple name and text changes reflecting the different data to be entered. The server-side servlet, this time mapped to the servlet path `/checkPhoneNumber`, is implemented as a class named `CheckPhoneNumberServlet`, and is shown in listing 6.7.

Listing 6.7 The CheckPhoneNumberServlet source code

```
package org.bibeault.sq.chap6;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class CheckPhoneNumberServlet extends HttpServlet {

    private static final String KEY_VALUE = "value";
    private static final String HEADER_VALID = "x-valid";

    public void doGet( HttpServletRequest request,
                      HttpServletResponse response )
        throws IOException {
        String value = request.getParameter( KEY_VALUE );
        StringBuilder digits = new StringBuilder();
        for (int n = 0; n < value.length(); n++) {
            if (Character.isDigit( value.charAt( n ) )) {
                digits.append( value.charAt( n ) );
            }
        }
        if (digits.length() == 10) {
            value = new StringBuilder()
                .append( '(' )
                .append( digits.substring( 0, 3 ) )
                .append( ')' )
                .append( digits.substring( 3, 6 ) )
                .append( '-' )
                .append( digits.substring( 6 ) )
                .toString();
            response.setHeader( HEADER_VALID, Boolean.TRUE.toString() );
        }
        else {
            response.setHeader( HEADER_VALID, Boolean.FALSE.toString() );
        }
        response.getWriter().write( value );
    }

    public void doPost( HttpServletRequest request,
                       HttpServletResponse response )
        throws IOException {
        doGet( request, response );
    }
}
```

In this servlet, the `doGet()` method obtains the value of the passed request parameter and iterates through it, collecting any digits that it finds in a `StringBuilder` instance named `digits`. If exactly 10 digits are found, a new string is constructed (using another instance of `StringBuilder`) placing the digits into the standard format, and returning them as the response body after setting the “x-valid” response head to `true`. If more or fewer than 10 digits are found, the header is set to `false` and the original value is returned unchanged. Again, `doPost()` is defined to perform the same action as `doGet()`.

This example shows us that the server-side save action resource can manipulate the value that will be placed in the `InPlaceEditor` in any way—presumably to make life easier for your users.

Now that you know how to use the `InPlaceEditor` for presenting and editing text, let’s look at another editor that *Scriptaculous* offers for when accepting a more restricted set of inputs from the user: the `InPlaceCollectionEditor`.

6.3 *The InPlaceCollectionEditor*

The `InPlaceEditor` allows us to place a seemingly display-only text element into an active edit mode so that the user is free to change the element’s value to anything he or she desires. But sometimes you may want to restrict the values that can be entered to a particular set of valid values. We could utilize a server-side validation resource, like the one we discussed in the previous section, to make sure that the user enters a valid value, but this essentially makes the user guess at what the valid values are, and it is a less-than-friendly way to approach the issue.

If we were using normal HTML form elements to present such a control, we’d probably employ either a set of radio buttons or a select element to let the user choose a single value from among a set of valid choices. The *Scriptaculous* set of controls doesn’t provide an in-place editor that displays radio-button style choices. However, it does offer a control that uses a select element while in active mode, in place of the text-input element that the `InPlaceEditor` uses.

This control is the `InPlaceCollectionEditor`. Our first step is creating one—we will find it to be similar in nature to its free-form-entry counterpart.

6.3.1 *Creating an InPlaceCollectionEditor*

In many respects, the `InPlaceCollectionEditor` is very similar to its text-based counterpart: a display-only target element is transformed into an active element that accepts user input. But rather than displaying a text box, a `<select>` element

(configured for single selection) is employed to present a predefined list of values to the user.

Unlike the `InPlaceEditor`, which had no set of predefined elements, the values for the `InPlaceCollectionEditor`'s select options are specified via a client-side list of values. A typical creation snippet for an instance of an `InPlaceCollectionEditor` might look like this:

```
new Ajax.InPlaceCollectionEditor(  
    'targetElement',  
    'doSomething.jsp',  
    {  
        okText: 'OK',  
        collection: [value1,value2,value3]  
    }  
);
```

This looks strikingly familiar to the creation of an `InPlaceEditor`, except for the new option named `collection`, which accepts an array whose elements serve as the options to the select element when the editor is placed into active mode. Omitting the `collection` option will not result in an error, but you'll get a less-than-useful select element with no options.

A simple example page, similar to the example in listing 6.2 for the `InPlaceEditor`, is shown in listing 6.8.

Listing 6.8 A simple `InPlaceCollectionEditor` example

```
<html>  
  <head>  
    <title>Simple InPlaceCollectionEditor Example</title>  
    <script type="text/javascript"  
      src="../../scripts/prototype.js"></script>  
    <script type="text/javascript"  
      src="../../scripts/scriptaculous.js"></script>  
    <script type="text/javascript">  
      window.onload = function() {  
        new Ajax.InPlaceCollectionEditor(  
          'theElement',  
          'transform.jsp',  
          {  
            okText: 'Upper me!',  
            cancelText: 'Never mind',  
            collection: ['one', 'two', 'three', 'four', 'five']  
          }  
        );  
      }  
    </script>  
  </head>
```

```
<body>
  <div id="theElement">Click me!</div>
</body>

</html>
```

This code is almost identical to the code of listing 6.2, except that we invoked the constructor for an `Ajax.InPlaceCollectionEditor` object and supplied a `collection` option providing five text strings as the collection choices. When this page is displayed and the editor is placed into active mode, we see a slightly different result than when we used an `InPlaceEditor`, as we can see in figure 6.8.



Figure 6.8
Simple `InPlaceCollectionEditor`
in active mode

Rather than the free-entry text box that was created when we used the `InPlaceEditor`, the `InPlaceCollectionEditor` creates an HTML `<select>` element. Expanding the element, as in figure 6.9, reveals that the choices we specified in the `collection` option have been used as the options for the select element.

As with the `InPlaceEditor` control, we have options! Let's take a look at them.



Figure 6.9
The `InPlaceCollectionEditor`
with the select
element opened

6.3.2 The *InPlaceCollectionEditor* Options

Aside from the addition of the `collection` option, the list of options for the *InPlaceCollectionEditor* is a subset of the options inherited from the *InPlaceEditor*. The options are listed in table 6.2.

Table 6.2 The *Ajax.InPlaceCollectionEditor* options

	Option	Description
Appearance options	<code>okButton</code>	A Boolean value indicating whether or not an OK button is to be displayed. Defaults to <code>true</code> .
	<code>okText</code>	The text to be placed on the OK button. Defaults to "ok".
	<code>cancelLink</code>	A Boolean value indicating whether a cancel link should be displayed. Defaults to <code>true</code> .
	<code>cancelText</code>	The text of the cancel link. Defaults to "cancel".
	<code>savingText</code>	A text string displayed as the value of the control while the save operation (the request initiated by clicking the OK button) is processing. Defaults to "Saving...".
	<code>clickToEditText</code>	The text string that appears as the control's tooltip upon mouse-over.
	<code>highlightcolor</code>	The color to be applied to the background of the text element upon mouse-over. Defaults to a pale yellow.
	<code>highlightendcolor</code>	The color to which the highlight color fades as an effect. Support for this option seems to be spotty in some browsers.
	<code>collection</code>	An array of items that are used to populate the select element.
CSS styling and DOM ID options	<code>savingClassName</code>	The CSS class name applied to the element while the save operation is in progress. This class is applied when the request to the saving URL is made, and it is removed when the response is returned. The default value is "inplaceeditor-saving".
	<code>formClassName</code>	The CSS class name applied to the form created to contain the editor element. Defaults to "inplaceeditor-form".
	<code>formId</code>	The ID applied to the form created to contain the editor element.
Callback options	<code>onComplete</code>	A JavaScript function that is called upon successful completion of the save request. The default function applies a highlight effect to the editor.
	<code>onFailure</code>	A JavaScript function that is called upon failure of the save request. The default function issues an alert showing the failure message.

Table 6.2 The `Ajax.InPlaceCollectionEditor` options (*continued*)

	Option	Description
Other options	<code>loadTextUrl</code>	The URL of a server-side resource to be contacted in order to load the initial value of the editor when it enters active mode. By default, no back-end load operation takes place and the initial value is the text of the target element. In order for this option to be meaningful, it must return one of the items provided in the <code>collection</code> option to be set as the initial value of the select element.
	<code>externalControl</code>	An element that is to serve as an external control that triggers placing the editor into active mode. This is useful if you want another button or element to trigger editing the control.
	<code>ajaxOptions</code>	A hash object that will be passed to the underlying Prototype Ajax object to use as its options hash.

If you compare table 6.2 with table 6.1 (which listed the options for the `InPlaceEditor`), you will see that the following options are missing here: `rows`, `cols`, `size`, `loadingText`, and `callback`. Seeing as the active element is a select element rather than a text field or text area, the omission of `rows`, `cols`, and `size` should not be surprising. Also, since there is no longer a text field in which to show any loading text, the `loadingText` option unnecessary, even though the `loadTextURL` option is still available to fetch a server-side value for the initial value of the select element. The omission of the `callback` option is a bit puzzling. The internal code for the `InPlaceCollectionEditor` overwrites any callback you specify with a function of its own (one that creates the query string for the save operation). This may be intentional, or it may be a bug in the version of Scriptaculous available when this chapter was written.

Use the `InPlaceCollectionEditor` Lab page (available from the chapter's control panel, or at chap6/inplace-collection/lab-inplace-collection.html) to test the effect these options have on the creation and operation of an instance of an `Ajax.InPlaceCollectionEditor`.

Next, we'll explore another class of control for helping users enter values into our applications.

6.4 The Ajax autocompleter control

The in-place editors gave us the ability to create controls that transformed display-only text into editable text, and those controls could tap into the power of the server. Another class of controls, the autocompleters, will help us make life easier for our users when they are faced with the dreaded situation of too much information!

Yes, it really is possible to have too much information—having the amount of information on the Web available at our fingertips is very useful, but it's easy to be overwhelmed with a deluge of data. When designing user interfaces, particularly those for web applications, which have the ability to access huge amounts of data, it is important to avoid flooding a user with too much data or too many choices. In this section and the next, we will explore Scriptaculous controls that help page authors present large lists of choices to users in a friendly and manageable fashion.

When presenting large data sets, such as report data, good user interfaces give the user tools they can employ to gather data in ways that are useful and helpful. For example, filters can be employed to weed out data that is not relevant to the user, and large sets of data can be paged so that they are presented in digestible chunks.

As an example, let's consider a data set that we'll be using throughout this chapter: a list of science fiction movies compiled from Internet sources. This data set consists of 1,460 titles. It's a large set of data, but still a small slice of larger sets of data (such as the list of all movies ever made, for example).

Suppose we wish to present this list to users so that they could pick their favorite sci-fi flick. We could set up an HTML `<select>` element that they could use to choose one title, but that would hardly be the friendliest thing to do. Most usability guidelines recommend presenting no more than a dozen or so choices to a user at a time, let alone many hundreds! And usability concerns aside, how practical is it to send such a large data set to the page each time it is accessed by potentially hundreds, thousands, or even millions of users on the web? Web application authors are often faced with such dilemmas.

In this section, we will explore the Ajax autocompleter control, and in the next, the local autocompleter control. The Ajax autocompleter (the `Ajax.AutoCompleteter` class) implements a control that utilizes Ajax mechanisms to present the user with a filtered list of choices from the server based upon a partial entry of the data. The Scriptaculous local autocompleter (the `Autocomplete.Local` class) performs a similar function, except that it uses data uploaded to the client rather than data stored on the server, and it may be more appropriate in situations involving a smaller data set that can be efficiently uploaded.

Let's get right to it.

6.4.1 Creating an Ajax autocompleter

The process of creating an Ajax Autocompleter instance with a class constructor specifying a few parameters and an options hash is a simple operation. However, there are some rather stringent requirements that must be met regarding the nature of the HTML elements that the Autocompleter manipulates, the response that the server returns, and even the CSS rules that must be defined in order for the control to operate reasonably.

The easiest part of the whole process is constructing the instance of the `Ajax.Autocompleter` class:

```
new Ajax.Autocompleter(
  'autoCompleteTextField',
  'autoCompleteMenu',
  'fixed-list.html',
  {
    minChars: 1
  }
);
```

The constructor accepts four parameters: the element name of (or a reference to) a text field that is to be populated with a data choice, the element name of (or a reference to) a `<div>` element to be used as a menu of choices by the control, the URL of the server-side resource that will supply the choices, and the usual options hash.

Listing 6.9 shows a simple example in a complete HTML page. You can launch this page from the control panel for this chapter's code examples, or open it directly at `autocompleter-ajax/simple-example.html`.

Listing 6.9 A simple example page for the `Ajax.Autocompleter` control

```
<html>
<head>
  <title>Simple Ajax Autocompleter Example</title>
  <script type="text/javascript"
    src="../scripts/prototype.js"></script>
  <script type="text/javascript"
    src="../scripts/scriptaculous.js"> </script>
  <script type="text/javascript">
    window.onload = function() {
      new Ajax.Autocompleter(
        'autoCompleteTextField',
        'autoCompleteMenu',
        'fixed-list.html',
        {}
      );
    }
  </script>
</head>
<body>
  <input type="text" value="Search" />
  <div id="autoCompleteMenu" style="display:none">
    <ul style="list-style-type:none">
      <li>1 Create the Autocompleter</li>
    </ul>
  </div>
</body>
</html>
```

```

    );
  }
</script>
</head>

<body>
  <div>
    <label>Text field:</label>
    <input type="text"
      id="autoCompleteTextField"/>
    <div id="autoCompleteMenu"></div>
  </div>
</body>

</html>

```

2 **Declare the text field**
3 **Declare the menu container**

This page creates an Ajax Auto-completer instance **1** in the `onload` function of the page. The parameters to the constructor reference a text field **2** and an empty `<div>` element **3**. It specifies a URL of “fixed-list.html” as its server-side resource, and specifies an empty options hash. When displayed, the page is rather less than exciting, as shown in figure 6.10.

But upon typing a character in the text box, we get some action going, as shown in figure 6.11. But what does the displayed list mean? And what does it have to do with the `x` character that was typed?

If you click on one of the listed choices, “Four” for example, you will note that the list that appeared when you typed a character into the text field goes away, and that the clicked text becomes the value of the text field. This list is the menu of auto-completion choices that the server-side resource returned when it was contacted by the control. The presented choices actually have nothing to do with

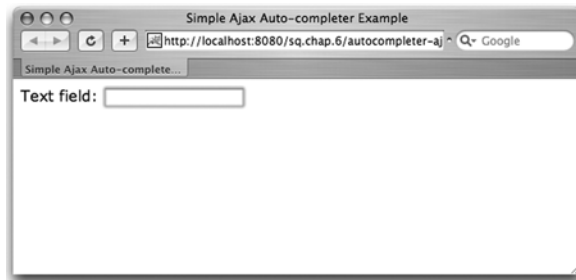


Figure 6.10
Initial state of the simple example page for the Ajax Auto-completer

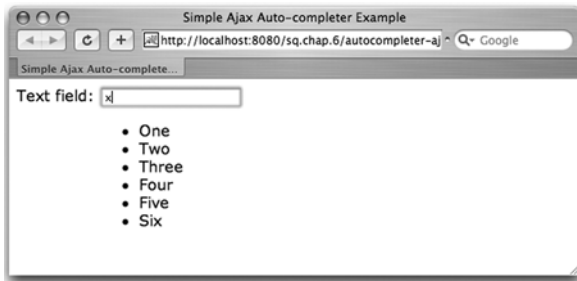


Figure 6.11
The simple example page for the Ajax Autocompleter showing choices

the x character—the fixed-list.html server resource returns (as its name implies) a fixed list, no matter what information it is fed, as shown in listing 6.10.

Listing 6.10 Brain-dead fixed-list server resource

```
<ul>
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
  <li>Four</li>
  <li>Five</li>
  <li>Six</li>
</ul>
```

This server resource doesn't seem very useful, and it isn't. But it does depict the format of the response the server-side resource is expected to return: an unordered list element populated with list item elements representing the choices the user can pick from to populate the text field. In order to be truly useful, the server-side resource needs to respond to the text already entered in the text field, using it to present a list of meaningful choices to the user. We will create just such a resource in section 6.4.3.

Another problem with this simple example is that when the menu of choices pops up, it just doesn't look, or even act, like a menu. It looks like, well, the unordered list that it is. Additionally, there is no feedback of any type indicating that clicking on one of the list items will do anything at all. These are grave usability problems that we will need to solve with CSS. More on that subject will be covered in section 6.4.4.

So far we've seen how we can construct the Autocompleter instance, and how we need to structure the HTML element that it will manipulate. Now let's take a look at the options.

6.4.2 Ajax.Autocompleter options

As was done for the in-place editor controls, a lab page has been set up for the Ajax.Autocompleter control, where you can play around with setting various options for the control. You can call up the page via the chapter's control panel, or find the page at `autocompleter-ajax/lab-autocompleter-ajax.jsp`.

The set of options available for the Ajax.Autocompleter control is shown in table 6.3.

Table 6.3 The Ajax.Autocompleter options

Option	Description
<code>paramName</code>	The name of the query parameter containing the content of the text field posted to the server-side resource. Defaults to the name of the text field.
<code>minChars</code>	The number of characters that must be entered before a server-side request for choices can be fired off. Defaults to 1.
<code>frequency</code>	The interval, in seconds, between internal checks to see whether a request to the server-side resource should be posted. Defaults to 0.4.
<code>indicator</code>	The ID or reference to an element to be displayed while a server-side request for choices is under way. If omitted, no element is revealed.
<code>parameters</code>	A text string containing extra query parameters to be passed to the server-side resource.
<code>updateElement</code>	A callback function to be triggered when the user selects one of the choices returned from the server. This function replaces the internal function that updates the text field with the chosen value.
<code>afterUpdateElement</code>	A callback function to be triggered after the <code>updateElement</code> function has been executed. By default, no function is triggered.
<code>tokens</code>	A single text string, or array of text strings, that indicate tokens to be used as delimiters to allow multiple elements to be entered into the text field, each of which can be autocompleted individually.

Let's look at how these options are used to tailor the Autocompleter control to our needs.

Specifying server-side parameters

The `paramName` option allows you to specify the name of the query parameter used to relay the contents of the text field when the Ajax request is sent to the server. The server resource uses the value of this parameter to filter the list of choices that it will return to the client.

Usually, the server resource will use whatever string is passed to it as a prefix filter to determine which choices to return (hence the “autocompleter” name for the control), but you can write the server resource to use the value in any way you deem fit. You can even ignore the input value completely, as our previous simple did, but that’s generally of little use. In the lab page for this control, the value of this parameter is hard-coded to “value”, as that is the parameter name expected by the server-side resource that provides the autocompletion list. (This resource is discussed in section 6.4.3.)

The `parameters` option can be used to pass extra query parameters to the server resource when a request for choices takes place. This string should be sets of name/value pairs, separated by ampersands, that follow the rules of querystrings. Be sure that any values containing non-alphanumeric characters are properly encoded using the `encodeURIComponent()` function. For example, if you wanted to pass three extra query parameters, the `parameters` option string could look like this:

```
one=1&two=2&three=3
```

The Ajax Autocompleter Lab page uses this option to pass any server delay setting that you might specify. (The purpose of this value is to artificially delay the server response to simulate a longer network latency than you get with a local server.)

Speaking of server-side latency, we want to have some control over how often and when the server gets hit by our control. Let’s see what options we have to help us with this.

Controlling when the autocompleter triggers

The `minChars` option allows you to specify the minimum number of characters that need to be entered into the text field before a server-side request can be made. Let’s say, for example, that we know that all or most of the choices begin with the word “The.” In such a case, you might wish to set the `minChars` options to 5, because until the fifth character is entered, all entries in the list would match and be returned as choices.

Internally, the control’s code checks at regular intervals to see whether a server request for more choices is appropriate. The `frequency` option allows you to override the default interval of 0.4 seconds. You might want to specify a smaller value if your server and connection are fast and can support a more frequent request load, or a larger value to slow it down if the default value hammers your server too quickly for it to keep up.

Now that we know how to keep the server happy, what about our users?

Keeping the user informed

The `indicator` option allows you to specify an HTML element on your page that is revealed while a server request for choices is being made, and that is hidden again when the request completes. This is most often used to display a message stating that a server operation has taken place, or to reveal an animation that indicates an active operation. This lets the user know that something is still happening when network latency delays the display of the choices.

The lab page for this control references a GIF animation that is revealed if you select the “Yes” radio button for the `indicator` option. You can set the `serverDelay` value to something like 3000 (3 seconds) to make sure that the request takes long enough for you to see the effect of this option.

Now let’s see how we can keep ourselves as well informed as the users of our pages.

Keeping your code informed

There are two options you can use as callbacks into your own code: `updateElement` and `afterUpdateElement`.

The `updateElement` option allows you to replace the function that will gain control when a choice is made from the available list presented to the user. That selection is passed as the single parameter to this callback function. Since this function replaces the internal function that updates the text field, that responsibility will fall on your callback function. This option is most useful when you want your page to do something other than update the text field.

If you want to let the control’s code update the text fields as normal, but also to be informed, you could register a callback with the `afterUpdateElement` option. This callback is invoked after the function identified by the `updateElement` option (be it yours or the internal function) has been executed. It is passed a reference to the text field as its first parameter, and the selected choice from the completion list is the second.

That’s all good stuff. Now let’s see what other capabilities the options can add to our control. For example, what if we would like the user to be able to enter multiple elements?

Allowing multiple completion items in the text field

You can enable the entry of multiple autocompleted items in the single text field attached to the `Ajax.Autocompleter` control by supplying the character (or characters) that you want to use to separate the entries as the value of the `tokens` option. This option should be either a string consisting of the single character you

want to use as a token (the comma is a popular choice), or a string array of single-character strings containing the characters to be used as delimiters.

For example, either of the following lines could be used:

```
tokens: ', ',  
tokens: [';', ','],
```

The latter example makes both the comma and the semicolon token characters. When a token character or characters are specified, entering the token character(s) at the end of the text field starts another, completely new autocompletion entry.

To demonstrate this capability, display the `Ajax.AutoCompleteLab` page, leaving all other options empty, but entering a single comma into the `tokens` field. Type “The” into the Movie Title text field. The autocompleter will list many movies starting with “The”. Pick a short one, and enter a comma at the end of the text field (which should now be populated with the short movie title that you picked) and type “The” again. The autocompletion menu reappears. Choose another short title and repeat. You can see that each entry separated by the token character (in this case, the comma) is treated as a separate choice.

That completes our tour of the Autocompleter options. Now let’s take a look at how we powered the back-end for the lab page we used to examine these options.

6.4.3 The sci-fi movie autocompleter servlet

If you’ve toyed around with the `Ajax.AutoCompleteLab` page at all, you will have noticed that when the autocompleter activates, it displays choices from a list of science fiction and horror movie titles. In this section, we will explore the server-side resources that provides this list. The pattern employed by this resource is suitable for most back-end elements that can power an autocompleter.

So far in this chapter, the active server-side resources we have employed have been fairly simple snippets of code suitable for small JSP pages. The autocompleter resource, on the other hand, is a bit more complex. Generally, autocompletion choices may come from text files on the server, or even from a database. Such complex operations are unsuitable for coding into a JSP page, so we will be utilizing the services of a Java servlet to perform the heavy lifting.

In fact, following modern web application best practices, the servlet will perform the processing necessary to create the list of items that match the autocompletion string passed to it, and will then forward the request on to a JSP page which will use those values to create the unordered HTML list. This pattern of performing processing in a servlet and using a JSP page to render the view is known

as the Model 2 Pattern. It follows the accepted practice of structuring web applications according to the MVC Pattern, at least to the extent that the limitations of the HTTP protocol allow.

Let's start by taking a peek at the servlet class.

The `MoviesAutoCompleterServlet` class

The controller portion of our autocompletion server-side resource is a servlet named `MoviesAutoCompleterServlet`. In the source code provided for this chapter, you will find the source of this servlet under the `WEB-INF/src` folder in the package hierarchy. Note that it is not customary to place the source code for web applications in the `WEB-INF` folder, or anywhere in the web application structure, for that matter; we have done so here as a convenience for you.

Data such as a list of movie titles would be kept in a database or in files on the filesystem. However, in order to keep these examples focused on the current subject matter, the servlet will obtain the list of possible choices from a string hardcoded into the program itself. Do not take this as an endorsement of this approach; it is used here only for demonstration purposes.

The source for the servlet is shown in listing 6.11.

Listing 6.11 The `MoviesAutoCompleterServlet` class

```
package org.bibeault.sq.chap6;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

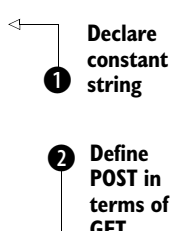
public class MoviesAutoCompleterServlet extends HttpServlet {

    public static final String KEY_VALUE = "value";
    public static final String KEY_DELAY = "delay";

    public static final String VAR_LIST = "list";

    public static final String VIEW_RESOURCE =
        "/autoCompleter-ajax/unordered-list.jsp";

    public void doPost( HttpServletRequest request,
                       HttpServletResponse response )
        throws IOException, ServletException {
```



1 Declare constant string

2 Define POST in terms of GET

```

doGet( request, response );
}

public void doGet( HttpServletRequest request,
                  HttpServletResponse response )
    throws IOException, ServletException {
    //
    // If a delay was requested, cause the delay to occur
    //
    if (request.getParameter( KEY_DELAY ) != null) {
        delay(request.getParameter( KEY_DELAY ));
    }
    //
    // Get the passed prefix and obtain the list of matches
    //
    String prefix = request.getParameter( KEY_VALUE );
    List<String> matches = findMatches( prefix );
    //
    // Set the matches as a scoped variable on the request
    // and shuffle off to the JSP
    request.setAttribute( VAR_LIST, matches );
    request.getRequestDispatcher( VIEW_RESOURCE )
        .forward( request, response );
}

private void delay( String delayValue ) {
    try {
        Thread.sleep( Integer.parseInt( delayValue ) );
    }
    catch (Exception e) {
        //On failure, perform no delay
    }
}

private List<String> findMatches( String prefix ) {
    List<String> matches = new ArrayList<String>();
    for (String choice : MoviesList.getMovies()) {
        if (choice.startsWith( prefix )) {
            matches.add( choice );
        }
    }
    return matches;
}
}
}

```

3 Declare the GET handler

4 Obtain delay value

5 Obtain prefix value

6 Set list as a request-scoped variable

7 Forward to the JSP page

8 Effect the delay

9 Declare method to find matches

This servlet, extended as usual from `HttpServlet`, is fairly simple in structure and operation. As it is generally considered poor practice to embed strings in code where they are difficult to find or reuse, the servlet first declares a number of constant strings ❶ for use in the code.

As we have no foreknowledge of whether the page author who will be using this servlet is going to specify a GET or POST operation, we support both by coding the `doPost()` method ❷ to simply call the `doGet()` method ❸.

Within the `doGet()` method, we first check to see if a `delay` parameter was passed ❹ as a query parameter (known in servlet parlance as a request parameter). If so, we pass that value to a private function ❺ that causes the thread running this code to go to sleep for the specified number of milliseconds. This delay is not a normal function that one would put into such a servlet. Rather, it exists only to help users of the Ajax.Autocompleter Lab page simulate longer network latencies than are normally experienced when serving data from a local server. Most readers of this chapter are probably serving their data locally, so this option is useful when checking how options like `indicator` act when network delays occur.

After causing any artificial delay, the servlet gets down to the real business at hand: building the list of movie titles that match the data passed as the `value` request parameter. The value of that parameter is obtained and is passed ❻ to a private method ❼ that creates a `List` of strings matching the prefix. It is this private method (the `findMatches()` method) that you would refactor in order to obtain the list from a more realistic data store, such as a database.

After the `List` is obtained, it is placed onto the request as a scoped variable named `list` ❽, and the request is forwarded to the JSP resource ❹ whose job is to render the list in an appropriate fashion before it is returned as the response. This JSP page is discussed in the next section.

The `findMatches()` method ❼ simply obtains the full list of movies from an abstract class named `MoviesList` (which we won't bother to show here, as it is simply a wrapper around a string array of movie titles—the source is included in the same folder as the servlet if you are curious). It then constructs the matching list from all titles that begin with the passed prefix.

As you can see, no knowledge of how the data is to be rendered as HTML is included in this servlet. It simply creates a `List` of the results and forwards it along to another component (a JSP page in this case) whose job it is to format the HTML. This is a good application of the principle of Separation of Concerns, and would allow this servlet to be reused to render the list in any format if we were to parameterize to which JSP the matching list were forwarded.

So how about that JSP page? How does it do its job?

The unordered-list-builder JSP page

When the autocompleter servlet we looked at in the previous section finished building its list of matching movie titles, it placed the list in a request-scoped variable named `list` and forwarded to a JSP page. That JSP page, `unordered-list.jsp`, is shown in listing 6.12.

Listing 6.12 The unordered-list-builder JSP page

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<ul>
  <c:forEach items="${list}" var="item">
    <li>${item}</li>
  </c:forEach>
</ul>
```

The purpose of this JSP page is to take a list of items and render an HTML `` element with the list items as the `` elements. To do this, it employs the services of the JSTL (JavaServer Pages Standard Tag Library) `<c:forEach>` action. The tag library for the JSTL core actions is declared, and the HTML construct for the unordered list is emitted by iterating over all elements in the scoped variable named `list`.

Note that this resource also exhibits the principle of Separation of Concerns. This JSP page could be used by any other server-side resource wishing to create an unordered HTML list out of a collection of elements. As long as the forwarding resource places the collection as a scoped variable named `list` (this requirement would be known as the page contract for this JSP page), this page will dutifully convert that collection into a `` construct.

Moreover, because we employed the JSTL `<c:forEach>` action, the exact nature of that scoped variable can be of many supported formats. We used a `List` implementation in our servlet, but other resources could have used a `Set`, a `Map`, or any other iterable member of the Java collection classes. It could even be a Java array. By keeping our controller and view resources decoupled in this fashion, each is more versatile, as well as easier to create, understand, and maintain. Ah, the joys of following best practices!

6.4.4 Styling the choices menu

You probably remember that when we ran the simple-example.html page for this control, the autocompletion menu looked pretty lousy. (Refer back to figure 6.11 if you need reminding.) If you subsequently ran the lab page for this control, you would have seen that the menu looked (and acted) much better (assuming you like burnt orange). The menu in the former page was unstyled, while the latter employed some CSS magic to make the menu look and act a lot better. In this section, we will take a look at some of the CSS rules we employed to achieve that. These rules can be found in the styles.css file in the same folder as the lab page.

One of the first elements we want to style is the unordered list itself, in order to make it look like a menu container. Since the `` element is contained within a `<div>` element with an ID of `autocompleteMenu`, the CSS selector `#autocompleteMenu ul` will cause the style rules to be applied to our unordered list while ignoring any other `` elements that might be on the page.

The rule is as follows:

```
#autocompleteMenu ul {
  border: 2px outset #cc9933;
  background-color: #cc9933;
  margin-left: 12px;
  padding: 4px 6px;
  width: 288px;
  height: 360px;
  overflow: auto;
}
```

We apply a border and background color to the element in order to make it look like a container. Its left margin is offset slightly from where it would normally appear, because we felt that made it look a bit better in relation to the autocompletion text field. Padding was added to keep the internal elements from looking too crowded. And finally, a fixed width and height were applied to constrain the size of the menu, along with an overflow rule that causes a scroll bar to be added to the menu if its contents exceed the constraining dimensions.

Now, on to styling the list items themselves. For that we used the following rule:

```
#autocompleteMenu li {
  color: white;
  list-style-type: none;
  padding: 0px;
  margin: 0px;
  border-bottom: 1px solid black;
  cursor: pointer;
}
```

We made the text color white and removed any bullets that might be shown on the items by setting the list style to none. All padding and margins were removed from the items so that they pack together tightly in the menu. In order to help visually separate the items from one another, a bottom border of a single pixel line was added. And finally, in order to clue the user in to the fact that the items are clickable elements, the cursor is set so that it changes to the little-hand pointer when the mouse pointer is over an item.

To add even more user feedback to the menu, we also added a `selected` rule, so that the item over which the cursor is positioned highlights itself:

```
#autocompleteMenu li.selected {
  color: #ffcc66;
  font-weight: bold;
}
```

This lightens the color of the item and makes it bold when the cursor passes over it.

The difference between the unstyled and styled versions of the menu can be seen in figure 6.12.

The Scriptaculous Ajax Autocompleter gives us a powerful server-assisted control. But what about cases where the server doesn't really need to be involved?

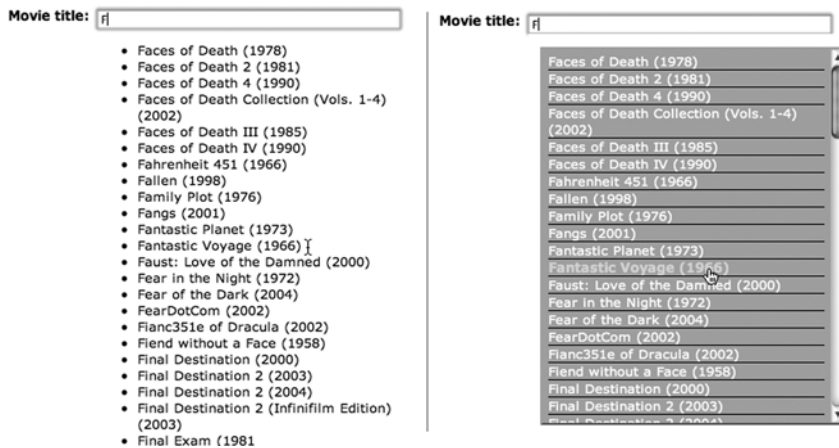


Figure 6.12 Lousy-looking menu vs. styled menu

6.5 The Scriptaculous local autocompleter control

One of the great features of the Ajax autocompleter control is that it allows you to access a large amount of data stored on the server in a fashion that is fairly seamless to a user of your application. The price paid for this access is, of course, network latency. Depending upon the nature of the data set, and the corresponding nature of the prefix entered by the user, a small or a large amount of data may need to be transferred. And that, of course, takes time.

There are times when you want to use a control similar in functionality to the Ajax autocompleter but with a backing data set that is small enough to upload to the client as part of the page load. For just such occasions, Scriptaculous makes available the local autocompleter control. As we will see, the local autocompleter shares a lot of characteristics with the Ajax autocompleter (and behind the scenes, a lot of code as well). But before just assuming that a local autocompleter will be faster than a remote one, be sure to test that supposition!

The local autocompleter must perform frequent calculations over what may still be a large data set, and in an environment that is usually not quite as swift as its server-side counterpart. You may find that there are occasions when any network delay experienced when using a remotely fed autocompleter may actually be faster than sorting through a large set of data on the client machine. So don't assume—test!

Once you have convinced yourself that a local control is the way to go for a particular situation, you'll need to create one. Let's see how.

6.5.1 Creating a local autocompleter

Creating a local autocompleter is almost identical to creating an Ajax autocompleter, which we did in section 6.4.1. The same type of constructor is used with almost the same parameters, and the HTML elements that the created autocompleter will manipulate must follow the same stringent rules. The major difference lies in how the backing data set used for autocompletion is identified to the control.

With an Ajax autocompleter, we supplied the URL of a server-side resource that would perform the necessary filtering, given the user input, and that would return only the data elements that matched. With a local autocompleter, we instead supply the full list of data elements as a JavaScript String array, and the control itself performs the filtering operation within its own client code. No server interaction whatsoever takes place (except for the initial loading of the page, of course).

A typical construction sequence might look like this:

```
new Autocompleter.Local(  
    'autoCompleteTextField',  
    'autoCompleteMenu',  
    ['abcdef', 'abcdeg', 'abcdfg', 'abcefg', 'abdefg', 'acdefg'],  
    {}  
);
```

We can see that, except for the third parameter, the parameters to the constructor are the same: the ID of the text field element, followed by the ID of the menu element, and with the options hash as the fourth parameter. For the third parameter, instead of the URL we used for the server-assisted autocompleter, we supply a small String array that contains all of the possible values. Generally, rather than typing in a large data set, such pages are generated with server-side templates (such as PHP or JSP) so that the array can be easily built into the page from server-side data.

The name of the object may seem a little odd: `Autocompleter.Local`. After seeing the `Ajax.Autocompleter`, I suppose one would expect the object to be named `Local.Autocompleter`, but for whatever reason, that's not the case.

An example of using the local autocompleter is shown in listing 6.13, and it can be found using the control panel for this chapter's code or by visiting the page at autocompleter-local/simple-example.html.

Listing 6.13 A simple example of a local autocompleter

```
<html>  
  <head>  
    <title>Simple Local Autocompleter Example</title>  
    <script type="text/javascript"  
      src="../scripts/prototype.js"></script>  
    <script type="text/javascript"  
      src="../scripts/scriptaculous.js"></script>  
    <script type="text/javascript">  
      window.onload = function() {  
        new Autocompleter.Local(  
          'autoCompleteTextField',  
          'autoCompleteMenu',  
          ['abcdef', 'abcdeg', 'abcdfg', 'abcefg', 'abdefg', 'acdefg'],  
          {}  
        );  
      }  
    </script>  
  </head>
```

```
<body>
  <div>
    <label>Text field:</label>
    <input type="text" id="autoCompleteTextField"/>
    <div id="autoCompleteMenu"></div>
  </div>
</body>

</html>
```

When the page is displayed and the character a is typed into the text box, the page shown in figure 6.13 will be displayed.

As you type successive letters in the alphabet, b, then c, then d, and so on, note how the available matching data is pared down to show only the items that match what you have typed into the text box, until you type the f, which uniquely identifies the item.

Also note the rather unsightly styling of the menu. We dealt with that for the Ajax auto-completer in section 6.4.4, and the exact same styling tips presented there also work for the local auto-completer. After all, the HTML elements are identical to those used for its Ajax counterpart.

You might expect that like all the other controls we've seen, the local auto-completer sports a number of options to adjust its behavior, and you'd be right. Let's take a look at them.

6.5.2 *Autocompleter.Local options*

The local auto-completer shares a great deal of code with the Ajax auto-completer and inherits all of the options that class supports. Please refer to table 6.3 in section 6.4.2 for details on those options, which work exactly the same way for a local auto-completer as they do for an Ajax auto-completer.

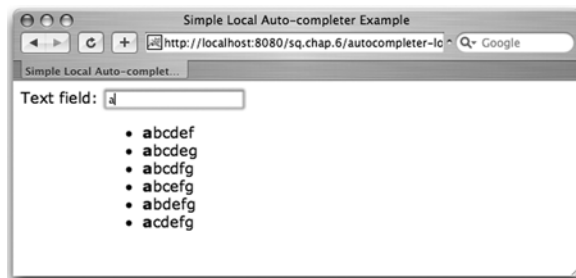


Figure 6.13
A simple example page for the local auto-completer

However, even though all the options may be inherited in code, it doesn't mean that they all make sense. The `paramName` and `parameters` options can be set, but because they make no sense in the context of a control that makes no server-side request, they are ignored. From the user point of view, the local autocompleter effectively does not support these two options.

The `indicator` option is another special case. This option is supported in code, and the element identified by the indicator is set to be displayed while the computations to determine the list are taking place. But because of the manner in which the browser and script engine interact, the element is hidden again before the browser ever gets a chance to actually show it. So while this option causes the appropriate code to execute, its effects are never seen, meaning that it is effectively not supported by the local control.

The local autocompleter does, however, sport a number of options that are not available for the Ajax autocompleter, and they are listed in table 6.4.

Table 6.4 Options for the `Autocompleter.Local` class

Option	Description
<code>choices</code>	The number of choices to display. Defaults to 10.
<code>partialSearch</code>	A Boolean option that enables or disables matching at the beginning of words embedded within the completion strings. Defaults to true (enabled).
<code>fullSearch</code>	A Boolean option that enables or disables matching anywhere within the completion strings. Defaults to false (disabled).
<code>partialChars</code>	The number of characters that must be typed before any partial matching is attempted. Defaults to 2.
<code>ignoreCase</code>	A Boolean option that specifies whether case is ignored when matching. Defaults to true.

In order to help you wrap your head around all these options, we created a lab page for the local autocompleter. You can access this page via this chapter code's control panel, or simply open the `autocompleter-local/lab-autocompleter-local.jsp` page.

This page allows you to manipulate the most common options for the local autocompleter and to specify the data set to be used for completion matching. You can either type your own data into the text area control, one entry per line, or you can click the Preload button. Clicking this button initiates an Ajax request that copies the movie titles you saw in the previous section into the text area.

When you click the Create Autocompleter button, an instance of `Autocompleter.Local` is created using the specified options and taking the contents of the text area (converted to a `String` array) as the completion data set.

Let's explore the options and customize how this control operates. You can follow along with the lab page loaded into your browser.

Controlling how many choices are shown

The `choices` option is a fairly straightforward setting. It limits the number of choices that are displayed in the menu. The default for this option seems a bit low at 10, so it is frequently overridden with a higher value. Specify a larger value (up to the size of the entire candidate set) if you want more choices to be shown.

To see this option in action, launch the lab page and populate the Choices List with the list of movie titles (by clicking the Preload button). Leave the Choices option field blank, and create the autocompleter by clicking the Create Autocompleter button. Type "The" into the autocompleter field and note that only 10 choices are shown, even though many more choices are available in the list.

Now set the Choices field to 100 and create a new autocompleter. Clear the autocompleter field and type "The" into the field again. Many more choices will be displayed.

Now let's see how we can control how the control performs matching.

Matching more than the beginnings of strings

The local autocompleter has four options that control how the matching algorithm is applied to the candidate data: `partialSearch`, `minChars`, `partialChars`, and `fullSearch`.

The `partialSearch` option tells the code to use the text in the autocompleter field to match the beginning of any word in the candidate strings. The `partialChars` option specifies the number of characters that must be typed before partial matching is attempted (not to be confused with `minChars`, which specifies the number of characters necessary before performing any matching at all).

Bring up the Local Autocompleter Lab page and leave all options blank except Choices. Enter a fairly high number for this option, such as 100, so that we can better see the results of the matching options. Preload the choices list with movie titles, and create the autocompleter. Type "day" (without the quotes, of course) into the autocompleter field. Note that because `partialSearch` is enabled by default, not only did we get matches such as "Day of the Triffids," which begins with the text we typed, candidates that include the word "day" or words that start with "day" are also included: "The Day the Earth Stood Still" and "Seven Days to Live."

Since `partialChars` defaults to 2, this partial matching was applied after we typed the two characters `d` and `a`. Change `partialChars` to 4, create a new auto-completer, clear the field, and type in “day” again. Notice that because we have not reached the threshold for partial matching to take place, only choices with “day” at the beginning of the candidate string are displayed.

When the `fullSearch` option is enabled the typed text is matched anywhere within the candidate string—not only at the beginning of the strings, and not only at the beginnings of words within the strings.

Set `partialChars` back to 2 (or clear its field), enable the `fullSearch` option, and create a new auto-completer. Clear the auto-complete field and type in “day” once again. Not only do we now get the set of choices we saw before (those that begin with “day” or contain words beginning with “day”), but we also get choices with “day” anywhere within the string, such as “Friday the 13th.”

Note that in order for `fullSearch` to take effect, `partialSearch` must also be enabled. If you specifically disable `partialSearch` (it is enabled by default), the `fullSearch` option is ignored.

What other options do we have for controlling matching? Let’s take a look.

Controlling the case sensitivity of matching

The matching that we’ve seen so far has all been case-sensitive. The `ignoreCase` option, when enabled (as it is by default), allows case-insensitive matching to take place.

Let’s see it in action. In the lab page, enable partial and full searching, and be sure that `ignoreCase` is enabled. Create an auto-completer and type “DAY” (all caps) into the text field. Note that matches are made regardless of the fact that the entry text was all uppercase.

Now, disable `ignoreCase`, create a new auto-completer, and type “day” into the text field. Note that the matching is now case-sensitive, and that only titles with the text “day” in all lowercase are displayed.

We now have two flavors of in-place editors and two flavors of auto-completers under our belts; we’ve broadened the scope of control types that we can add to our web application pages. Next, let’s take a look at yet another Scriptaculous control for our toolbox.

6.6 The slider control

So far we have seen two types of Scriptaculous controls that close the gap between desktop applications and web applications: the in-place editor and the autocompleter. The in-place editor is useful for switching between display-only and editable versions of data, and the autocompleter is great for presenting users with filtered lists of choices that match text they are typing into a text field. But another powerful desktop control that is not available to web developers as a native control is the slider, which allows users to choose a single value from a range.

A slider control is usually presented as a track that represents the range of values available, and a handle that represents the selected value within that range. Figure 6.14 shows some representative slider controls from various desktop programs:

- (a) The volume control from Apple's iTunes
- (b) The screen resolution control from the Windows XP Display Options Control Panel
- (c) The Mac OS X System Preferences dock size control
- (d) The volume control from the Windows XP Sound Panel

Note that sliders can be displayed in either a horizontal or vertical orientation. When horizontal, the left end of the track usually represents the lowest value, while in a vertical orientation, the bottom of the slide is usually the lowest value. The slider is controlled by “grabbing” the handle with the mouse and dragging it to the desired position within the track. It can also be manipulated by clicking in the track itself, which moves the handle toward or exactly to the location of the click.

The slider is a powerful control in that it allows users to select from a range of values without needing to type values into a text field, and without the possibility

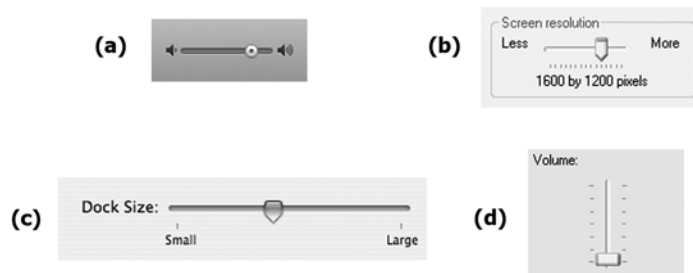


Figure 6.14
Sliders of various types
from desktop applications

of a validation error. Because HTML provides no slider control for the Web, Scriptaculous provides one for you to use on your pages.

6.6.1 **Creating a slider control**

A slider control is created by constructing an object and providing parameters and an options hash, as usual. In the case of the slider control, the object class is `Control.Slider`, and a typical construction statement might look like this:

```
new Control.Slider(  
  'sliderHandle',  
  'sliderTrack',  
  {}  
);
```

The first parameter must be the ID of (or element reference to) the block element serving as the handle of the slider, and the second parameter is the ID of (or element reference to) the track. The third parameter is the ubiquitous options hash.

As with the other controls, the HTML elements to which the slider will be attached must follow certain rules. In the case of the slider there are two elements: the track element, which must be a block element (most often a `<div>` element, but we'll see the use of another element type in just a little while), and the handle element, which should be contained within that track element. A typical HTML structure for the slider control is as follows:

```
<div id="sliderTrack">  
  <div id="sliderHandle"></div>  
</div>
```

Placed on a page with no styling, you wouldn't see much, as the `<div>` elements have nothing to display, and the `Control.Slider` doesn't do anything about that. It is the responsibility of the page author to apply styling to the `<div>` elements to achieve the desired look for the control. When `<div>` elements are used, as in the previous example, width and height CSS attributes, along with borders, background colors, and similar rendition attributes, are applied to achieve the visual aspects of the control.

Let's take a look at a page that uses the preceding snippets to create a functional slider control. The code for this page can be accessed via the Simple Example link in the Slider section of the control panel for this chapter's code examples, or within the file `slider/simple-example.html`. The code is shown in listing 6.14.

Listing 6.14 A simple Slider control example

```
<html>
  <head>
    <title>Simple Slider Example</title>
    <script type="text/javascript"
      src="../scripts/prototype.js"></script>
    <script type="text/javascript"
      src="../scripts/scriptaculous.js"></script>
    <script type="text/javascript">
      window.onload = function() {
        new Control.Slider(
          'sliderHandle',
          'sliderTrack',
          {}
        );
      }
    </script>
    <style type="text/css">
      #sliderTrack {
        width: 175px;
        height: 12px;
        background-color: white;
        border: 1px blue solid;
        cursor: pointer;
      }
      #sliderHandle {
        width: 4px;
        height: 12px;
        background-color: green;
        cursor: pointer;
      }
    </style>
  </head>

  <body>
    <div id="sliderTrack">
      <div id="sliderHandle"></div>
    </div>
  </body>

</html>
```

Note that the CSS styles we applied in this example create simple box elements. The track is a hollow blue-outlined box completely containing the handle, while the handle is a solid green box. We also added the pointer cursor to these elements, so that the user can see that these are active elements.

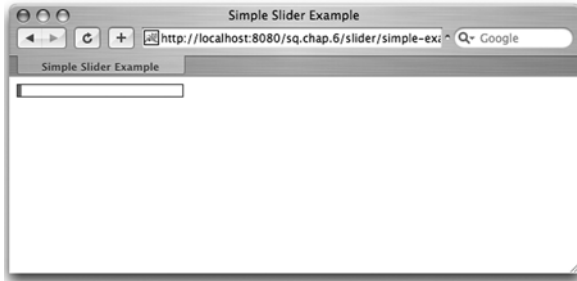


Figure 6.15
A simple Slider control
example

When displayed, this page appears as shown in figure 6.15. It's not all that exciting, but it is a working slider control, if not a particularly useful one.

With the page displayed, click anywhere within the track element. Note how the handle jumps to the clicked location, as shown in figure 6.16. This change in location also changes the value represented by the control. Just what that value is, we have no way of knowing at this point, but we'll rectify that in a later example.

Display the page again (if you closed it) and, this time, click and drag on the handle. Note how the handle follows the movement of the cursor until you let go of the mouse button.

Even though we used some very simple styling in this example, you have a lot of leeway in how you style sliders. By simply changing the CSS rule for the track a bit, as follows, we can change the visual appearance of the slider to what is shown in figure 6.17.

```
#sliderTrack {  
  width: 175px;  
  height: 4px;  
  background-color: pink;  
  cursor: pointer;  
}
```

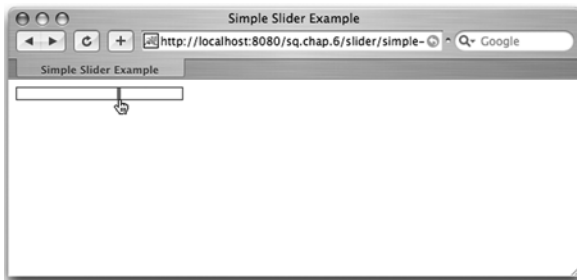


Figure 6.16
Slider handle moves to
a clicked location on the track

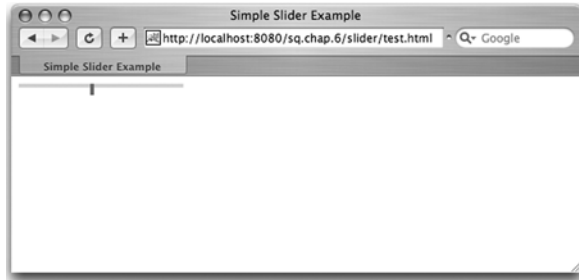


Figure 6.17
A Slider control with a narrow track

This is usable, but perhaps a trifle austere. In the next section, we'll see if we can spiff it up a bit by employing images.

6.6.2 Using images with a slider

The desktop examples of slider controls that were presented in figure 6.14 looked a bit nicer than the sliders we created in the previous section using simple blocks of color. To spruce up the control, we'll use some images for the track and the handle.

Using the control panel for this chapter's code, click on the Image Example link in the Slider section. The page shown in figure 6.18 will be displayed. Now that's more like it!

Implementing this change to the slider (which is obviously modeled after slider *c* in figure 6.14) is surprisingly simple. The code for this page, available in `slider/image-example.html`, is almost a carbon copy of the simple example page we examined in listing 6.14. The new page is shown in listing 6.15, with the differences from the simple example page highlighted in bold.

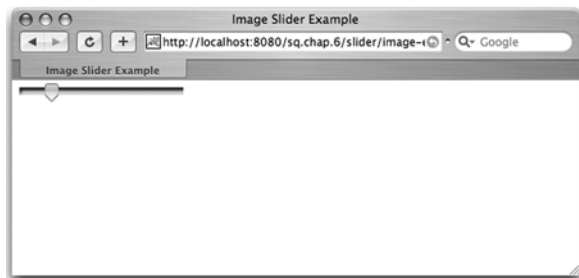


Figure 6.18
A Slider control constructed using images

Listing 6.15 The Image Slider Example page

```

<html>
  <head>
    <title>Image Slider Example</title>
    <script type="text/javascript"
      src="../scripts/prototype.js"></script>
    <script type="text/javascript"
      src="../scripts/scriptaculous.js"></script>
    <script type="text/javascript">
      window.onload = function() {
        new Control.Slider(
          'sliderHandle',
          'sliderTrack',
          {}
        );
      }
    </script>
    <style type="text/css">
      #sliderTrack {
        width: 175px;
        height: 10px; ① Adjust the appearance of the track
        background-image: url(track.gif);
        background-repeat: no-repeat;
        cursor: pointer;
      }
      #sliderHandle {
        margin-top: -4px; ② Shift the location of the handle
        cursor: pointer;
      }
    </style>
  </head>

  <body>
    <div id="sliderTrack">
      
    </div>
  </body>

</html>

```

The first change (other than the page title) rearranges the track styling ① to have a slightly different size (the height of the track image), and to have a nonrepeating background image that resembles a grooved track. Even though we've made the <div> element the same size as the background image, the no-repeat attribute is necessary, as some browsers (such as Internet Explorer) will expand the size of

the `<div>` to accommodate its contents, and we'd see the track image tiled across its background. We can't make the track an `` element, since it needs to contain the handle element, and image elements aren't containers. So the background image trick will work just as well.

The next change was to the CSS rule for the handle ❷. We removed the dimension attributes (as the handle is no longer a `<div>`, as we'll see next) and used a negative margin value to nudge the top of the handle up a bit in relation to the track. The handle element itself was changed from a `<div>` to an image element ❸ that references the image for the pentagonal handle.

The green version of the handle was used for this example. Within the slider folder you will find handle images in numerous colors, as well as a gray one useful for indicating when the control is disabled.

This isn't the only construct that would work. If you'd rather use two images instead of using the background image trick, you could place the images within a relatively positioned parent, and use absolute positioning to place the image elements in the correct locations with respect to the parent and each other. Which construct (or any possible others) you choose depends upon the context in which the control will appear, as well as your own whims.

That is all very nice, but we still don't have a control that's all that useful. In section 6.6.4, we will explore a more in-depth example that shows the slider in a more useful light, but first we'll take a look at some of the options available to a Slider control.

6.6.3 The Control.Slider options

As we write this, the documentation for the Slider control on the Scriptaculous site is rather seriously out of sync with what is actually implemented and working in the code. In this section, we discuss only the options that are known to be supported and that work correctly with the version of the Slider control upon which this book is based. The options for the Control.Slider control are shown in table 6.5.

Table 6.5 The Control.Slider options

Option	Description
axis	The orientation of the control, which can be set to <code>horizontal</code> or <code>vertical</code> . The default orientation is <code>horizontal</code> .
range	The range of the slider values, defined as an instance of a Prototype ObjectRange instance. Defaults to 0 through 1.

Table 6.5 The `Control.Slider` options (*continued*)

Option	Description
<code>values</code>	The discrete set of values that the slider can acquire. If omitted, all values within the range can be set.
<code>sliderValue</code>	The initial value of the slider. If omitted, the value represented by the left-most (or top-most) edge of the slider is the initial value.
<code>disabled</code>	A Boolean value that specifies whether the slide is initially disabled. Defaults to <code>false</code> (not disabled).
<code>onChange</code>	Callback invoked when the slider value changes.
<code>onSlide</code>	Callback invoked when the handle is being dragged.

As with the other controls, a lab page has been set up where you can play around with various settings for the most common Slider options. You can open the page from the control panel for this chapter's code, or simply open the `slider/lab-slider.html` file.

This lab is a bit different in some ways from those of the other controls. In most of the other lab pages, values that you enter into the lab page become direct properties of the options hash for the control. In the Slider Lab page, a bit of interpretation or extra processing is needed (for example, the minimum and maximum values you specify need to be converted to an `ObjectRange` object rather than being directly placed into the options hash). The Applied Options section of the lab page is much more important here, as it will show you what options were actually created and passed to the control's constructor.

With that said, let's dig in.

Setting the orientation

The `axis` option can be used to specify a horizontal or vertical orientation for the Slider control. As we have seen in our examples so far, the control defaults to a horizontal orientation.

In a horizontal control, it's usually natural to have the lowest value at the left end of the control, and the highest value at the right end. But in a vertical control, depending upon the context, it can make sense either way, with the highest values at the top of the control, or at the bottom. The manner in which the values are assigned is covered in the next section.

Note that the orientation of the axis does not in any way affect the appearance or dimensions of the control. When placing a Slider control in a vertical orientation, just be sure to specify your CSS dimensions (or images) to create a tall narrow control rather than a long, low control, as would be natural for the horizontal orientation.

When you select a vertical orientation in the Control.Slider Lab page, not only is the axis option added to the options hash, but the CSS rules that apply to the control element are also changed to use alternate styling and images to create a tall control. It is your responsibility as a page author to ensure that the appearance of the HTML elements attached to the control match its orientation.

When creating a slider control in either orientation, we'll obviously want to indicate the value range it represents. Let's see the options that allow us to do just that.

Setting the range of values

Two options, `range` and `values`, set the range of values that the slider can acquire.

The `range` option specifies a pair of values that represent the extreme ends of the control. When in horizontal orientation, the first value is assigned to the left-most edge of the slider, and the second value to the right-most edge. In vertical orientation, the first value is assigned to the top of the control, and the second value to the bottom.

This option is expressed as an instance of a Prototype `ObjectRange` class. You can either create such an instance the "normal" way, like this,

```
new ObjectRange(value1,value2)
```

or you can employ the handy `$R()` Prototype helper function to create a range using this shortened notation:

```
$R(value1,value2)
```

If we wanted to set up a slider such that the range of values spans from 1 at the left end of the control to 100 at the right end, we would specify the `range` option as follows:

```
range: $R(1,100)
```

If we could see the values that the slider was acquiring (patience, that's coming up next), we'd notice that the slider acquires not only integer values, but also all real values within the range. If we move the handle to about what we think is the middle of a control with a range of 1 through 100, we might expect the value to be 50, but instead we may find that the value is 48.7573572347. If we want to specify the

discrete values within the range that the slider should acquire, we can do that with the `values` option.

The `values` option specifies an array of the values that the slider can be set to. Interesting (but not always pleasing) behavior can be achieved by making the `values` and `range` options not match each other, but usually you want to make sure that they are in sync. For example, if we set the range to 1 through 100, as shown in the previous code examples, and specified the `values` option as follows,

```
values: [1,25,50,75,100]
```

the slider can only be set to the discrete values listed as the handle is moved.

If we wanted to allow the slider to achieve all integer values within the range, we'd construct an array containing all those values to pass as the `values` option. The `Control.Slider Lab` page demonstrates this when you choose to enable the `Discrete` option. When `Discrete` is true, all integer values within the specified range are added to an array used as the value of the `values` option.

At any time, the value of the slider can be set programmatically by calling the `setValue()` method of the `Slider` instance:

```
sliderInstance.setValue(47);
```

If a `values` list was provided, and the value specified in the `setValue()` method is not one of the discrete values in the list, the value will be rounded to the nearest permissible value.

If, for some reason, we wanted to reverse the direction and have the control span from 100 down to 1, we would simply use a `range` option like this,

```
range: $R(100,1)
```

and specify a `values` option with valid decreasing values. Omitting the `values` option when using a reversed range seems to put the control in an odd state, so be sure to use the `values` option when using a reversed range.

Next, we'll look at toggling between an enabled and a disabled state. Many of the HTML form controls exhibit such an ability, and the slider is no exception.

Enabling and disabling the control

By default, a `Slider` instance is created in the enabled state. However, you can start the slider in a disabled state by specifying this option:

```
disabled: true
```

You can also affect the state of the option programmatically with the `setDisabled()` and `setEnabled()` methods of the `Slider` instance. Note that these methods will

have no effect on the appearance of the control. If you want a visual difference between states, you will need to modify the CSS attributes of the Slider elements accordingly. The Control.Slider Lab page shows an example of such machinations.

It's all well and good to be able to configure the slider with the options we've seen so far. But we also need to know what's going on once the slider is put into operation. Let's see how.

Receiving notification of value changes

A slider whose value is unknown is a poor control indeed. Two callback options allow you to be notified when the value of the slider changes, and to obtain the specific value that the slider has acquired. These callbacks are most often used to cause the current value of the slider to be displayed to the user. The lab page uses these callbacks for this purpose, and the extended example of section 6.6.4 will show how this can be achieved.

The `onChange` callback is invoked whenever the value of the slider changes, and its single parameter contains the current value of the slider. It is even invoked when the value is changed by the control's `setValue()` method.

The `onSlide` callback is invoked, also with the current value as its parameter, whenever the handle is being dragged along the track.

Either or both of these callbacks are most often used to keep a display of the slider's value in sync with the internal value of the control, but they can be used for any other purpose related to the current value of the control (such as adjusting the size of chart elements).

Now that we know the range of options available to us, let's use that knowledge in a more in-depth example.

6.6.4 A more absorbing example

The simple example in listing 6.14 showed us a working slider, but without a display of the slider's value, it wasn't a very useful control. In this section, we will examine a page with a slider control that not only displays its current value, but also can visually change its appearance when toggled between enabled and disabled states. This page builds upon the pages we saw in listings 6.14 and 6.15, and it can be found in the `slider/extended-example.html` file, or you can display it from the chapter code's control panel.

When initially displayed, the page will appear as shown in figure 6.19.

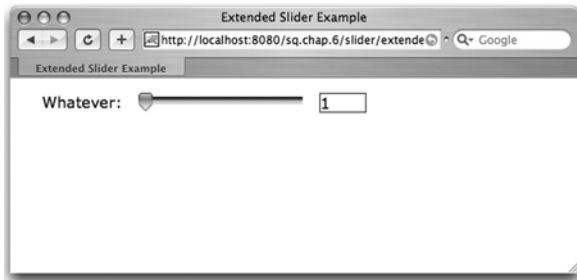


Figure 6.19
Initial state of
the Extended Slider
Example page

This page presents a slide constructed using images that are labeled, and it shows its current value to its right. If you modify the value of the slider by dragging the handle or clicking in the track, the numerical display changes to reflect the new value, as shown in figure 6.20.

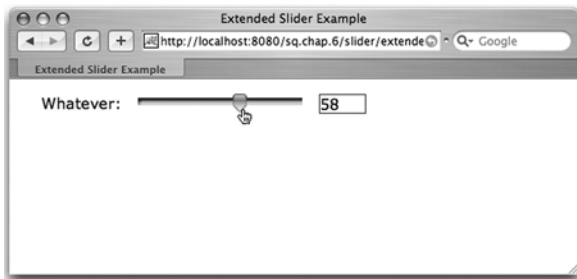


Figure 6.20
The extended example with a
new value

The complete listing for this page is shown in listing 6.16, and its important aspects will be discussed in the following subsections:

Listing 6.16 Code for the Extended Slider Example page

```

<html>
  <head>
    <title>Extended Slider Example</title>
    <script type="text/javascript"
      src="../scripts/prototype.js"></script>
    <script type="text/javascript"
      src="../scripts/scriptaculous.js"></script>
    <script type="text/javascript">
      window.onload = function() { 1 Create slider
        createSlider(1,100); in onload
      }

      function createSlider(minValue,maxValue) { 2 Generate
        var values = []; discrete values array

```

```

for (var n = minValue; n <= maxValue; n++) values.push(n);
new Control.Slider(
  'sliderHandle',
  'sliderTrack',
  {
    range: $R(minValue,maxValue),
    values: values,
    sliderValue: minValue,
    onChange: function(value) {
      $('sliderValue').innerHTML = value;
    },
    onSlide: function(value) {
      $('sliderValue').innerHTML = value;
    }
  }
);
$('sliderValue').innerHTML = minValue;
}
</script>
<style type="text/css">
body {
  margin: 16px;
}
#sliderTrack {
  background-image: url(track.gif);
  background-repeat: no-repeat;
  background-position: 0px 4px;
  width: 175px;
  height: 10px;
  cursor: pointer;
}
#sliderHandle {
  cursor: pointer;
}
#controlContainer label,
#controlContainer #sliderTrack,
#controlContainer #sliderValue {
  float: left;
  margin-left: 18px;
}
#sliderValue {
  border: 1px solid maroon;
  width: 26px;
  font-size: 88%;
  padding: 3px 6px;
}
</style>
</head>

<body>

```

3 Construct slider control

4 Define callback closures

5 Initialize value display

6 Apply track styles

7 Apply handle styles

8 Line up elements

```

<div id="controlContainer">
  <label>Whatever:</label>
  <span id="sliderElements">
    <div id="sliderTrack">
      
    </div>
  </span>
  <span id="sliderValue"></span>
</div>
</body>

</html>
```

9 Declare HTML elements

The following sections examine the important aspects of this example, referring back to the code as necessary.

Creating and styling the HTML elements

In our previous examples, we've seen the simple HTML elements that can be used to represent the track and the handle of the slider control. In this example, we've used a construct similar to the image example (shown earlier in listing 6.15) in that the track is a `<div>` element with a background image, and the handle is an image element.

We've also defined a sort of uber-control in this example, where the label and the value display are both considered elements of the slider, along with the track and handle. To this end, the HTML elements that define the uber-control are a bit more complicated than what we have seen so far, but not by much.

In order to contain the elements that make up the control, we created a `<div>` element named `controlContainer` **9**. Obviously, if we're going to be creating more than one of these on a page, we're going to have to be a lot smarter in our naming. But this is just a simple example. Once we get a handle on how to get our uber-slider working on this test page, we would want to create a JavaScript object out of it that would handle all the details and ensure that generated IDs are unique.

Within `controlContainer` we placed the control elements: the `<label>` element, a `` to contain the track `<div>` and handle image named `sliderElements`, and a `` to serve as the value display named `sliderValue`.

That's all pretty straightforward, but if we were to display that in a browser without any styling, we wouldn't be thrilled with the result. The styling applied to the track **6** and handle **7** should seem familiar by now. The track is given a background image and a pointer cursor, and the handle, already an image, is given only the active cursor.

However, unstyled, the various elements of the uber-control will not lay out in a line. In order to make the elements line up as we expect, we applied a style **8** to them that forces them to float left rather than to wrap to another line in the browser. We also gave them a little margin to prevent any bunching up.

With the elements created, we're ready to create the `Control.Slider` instance.

Setting up the slider's range

The instance of `Control.Slider`, which we'll hook up to the track and handle we created in the previous section, is created in the `createSlider()` function, which we invoke from the page's `onload` handler **1**. The `createSlider()` function accepts two parameters that specify the minimum (left-most) and maximum (right-most) values for the slider. No checks are made to ensure that `minValue` is less than `maxValue`, but that's a check we'd want to add (or to add code to account for) in a real-world implementation.

The minimum and maximum values are used to create the `ObjectRange` instance that will be used as the slider's `range` option, and they are also used to populate an array **2** of integer values that will be used as the slider's `values` option.

The slide is constructed **3** using these values and specifying the slider's initial value as `minValue`. That's actually the default, but since it's an assumption made by other elements on the page, we felt it was best to be explicit about it.

Now that the slider's range and value are set up, we want to make sure that the slider's current value will always be displayed in the `` element with the ID of `sliderValue`.

Displaying the slider's value

Recall that we set up a `` named `sliderValue` that will display the current value of the slider. In order to cause the value to be displayed in this element whenever the slider's handle is moved or its track is clicked (or even if some code calls the slider's `setValue()` method), we set up two callback closures in the options hash of the `Control.Slider` constructor **4**. The `onChange` and `onSlide` options are set up to perform the same task of placing the passed value into the `sliderValue` element.

Since these closures perform the same function, it might have been wiser to implement a named function that could be referenced by both options. Certainly if the code were any more complex than that shown here, repeating it twice would be a poor thing to do.

These closures will display the value of the slider once its value has been changed. In order to show the value when the page is initially displayed, the initial value is set into the `sliderValue` element right after the constructor completes ❸.

And that's it! You should now have enough knowledge at your fingertips to go about creating a JavaScript object class for the uber-slider. Once you have finished this book, particularly the Prototype sections on extending and creating object classes, actually implementing such an uber-slider control would be a great exercise to ensure that you've thoroughly understood the material.

6.7 Summary

In this chapter, we've examined some controls that Scriptaculous makes available to us as page authors. These controls help give web application developers some of the power that is enjoyed by their desktop application counterparts.

We saw a set of two in-place editor controls that transformed static display values into active elements for editing. The `InPlaceEditor` utilizes text fields and text areas in order to give the user the ability to enter free-form text, and the `InPlaceCollectionEditor` utilizes a single-selection select element to give the user a list of finite choices. Both of these controls employ Ajax technology to allow for assistance from server-side resources, be it for validation, reformatting, model updates, or whatever other server-side activity is appropriate.

We also discussed two autocompleter controls that present the user with a list of filtered choices based upon the text that they have already entered into a text box. Though differing in some key aspects, these controls are reminiscent of the combo box control available for desktop applications; they present a drop-down list where users can either choose from the list or type their own nonlisted value. The two autocompleter controls differ in that one obtains its filtered list from the server (the Ajax autocompleter) and the other filters a list of all candidate values on the client (the Scriptaculous local autocompleter).

And lastly, we examined a slider control that mimics its desktop counterpart, allowing users to select one value from within a predefined range. These controls not only give you some prepackaged extended controls to use within your web application's pages, they can serve as inspirations to create your own controls that perform functions that might be difficult or awkward using the predefined set of HTML form controls. Just as Scriptaculous uses the power of Prototype to easily create such controls in an object-oriented fashion and with a minimum of fuss and bother, so can we.

In the next chapter, we'll take a look at another feature that Scriptaculous brings to the party in order to help close the gap between desktop and web applications: drag and drop.

Prototype & Scriptaculous IN ACTION

Dave Crane and Bear Bibeault with Tom Locke

Common Ajax tasks should be easy, and with Prototype and Scriptaculous they are. Prototype and Scriptaculous are libraries of reusable JavaScript code that simplify Ajax development. Prototype provides helpful methods and objects that extend JavaScript in a safe, consistent way. Its clever Ajax request model simplifies cross-browser development. Scriptaculous, which is based on Prototype, offers handy pre-fabricated widgets for rich UI development.

Prototype and Scriptaculous in Action is a comprehensive, practical guide that walks you feature-by-feature through the two libraries. First, you'll use Scriptaculous to make easy but powerful UI improvements. Then you'll dig into Prototype's elegant and sparse syntax. See how a few characters of Prototype code can save a dozen lines of JavaScript. By applying these techniques, you can concentrate on the function and flow of your application instead of the coding details. This book is written for web developers with a working knowledge of JavaScript.

What's Inside

- Explore Prototype's Ajax helper classes
- How to add Scriptaculous effects and controls
- Closures in JavaScript
- Over 100 working examples

Dave Crane is a UK-based Ajax expert and coauthor of the best-selling *Ajax in Action* as well as the follow-up, *Ajax in Practice*.

Bear Bibeault is a US-based Java developer and coauthor of *Ajax in Practice*. **Tom Locke**, creator of Hobo for Rails, contributed the coverage of Rails in this book.

“Shows how Prototype and Scriptaculous let you concentrate on what's important: implementing your ideas.”

—Thomas Fuchs
Creator of Scriptaculous
from the *Foreword*

“Of all the books on my shelf, this is the one I go to the most.”

—Philip Hallstrom
CardPlayer.com

“Simplifies Ajax development—a great reference.”

—Mark Eagle
MATRIX Resources, Inc.

“Can't wait to implement ideas from this book!”

—Jeff Cunningham
The Weather Channel
Interactive

www.manning.com/crane3

ISBN-10: 1-933988-03-7
ISBN-13: 978-1-933988-03-0



9 781933 988030