Extracted from:

Pragmatic Ajax

A Web 2.0 Primer

This PDF file contains pages extracted from Pragmatic Ajax, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragmaticprogrammer.com.

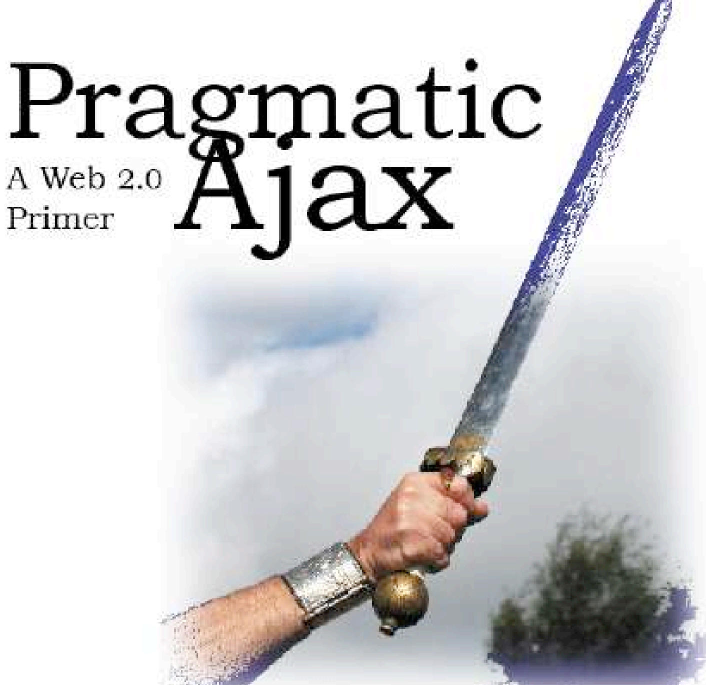
Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2005 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Programmers



Justin Gehtland
Ben Galbraith
Dion Almaer

Chapter 7

Ajax UI, Part II

In the previous chapter, we started to look at using some of the available Ajax JavaScript libraries to drive the user interface in a browser. Understanding how these libraries help you more efficiently control the UI is Step 1. Step 2 is understanding what you should do with your newfound tools.

This chapter will present some of the standard techniques for utilizing Ajax on the UI. We'll talk about validation, notification, and data management strategies that have proven they increase the utility and usability of web applications. Later, we'll talk about some antipatterns, too, the things you should avoid and the tests you should apply when Ajaxifying your application. This chapter isn't an exhaustive treatise. Our intent is to give you a set of foundational tools for deciding how to (and when not to) proceed.

7.1 Some Standard Usages

Let's look at several common applications of Ajax using the libraries we talked about in the previous chapter: Prototype, Script.aculo.us, and Dojo.

Server-Side Validation

Web applications face a variety of standard problems. Validation is one that has spawned an infinite array of potential solutions. We have learned over time that there is one universal delineation to be taken into account: the server side versus the client side. Or so we thought. Client-side validation is handy for our users because they get "instant" feedback about the correctness of their data entry without having to

wait for the whole page to refresh. Client-side validation is largely useless to the application developer, however, since it is trivial for a user to circumvent client-side JavaScript. Heck, users can ignore our rendered HTML entirely and craft requests to our system using Telnet. Therefore, server-side validation is always mandatory. Client-side validation is a usability enhancement for our users.

Ajax allows us to combine the two techniques for greater usability. The problem with client-side techniques is that the validation rule itself has to be portable to the browser. This means you can execute regular expression matches, required field checking, and even small-scale data comparisons (for example, is the state abbreviation one of the standard 50 two-letter abbreviations?). You can't, however, validate the inputs against your database or against any server-side resident data or rules. With Ajax, we get the benefits of client-side validation ("instant" feedback without a page refresh) but the power of server-side validation (comparison against server-resident data or rules).

This means we can create web applications with full validation the way we have historically been able to do only in *fat client* applications. We can use a full-fledged rules engine, for example, for validating individual data fields. But keep in mind that we are still required to re-validate the data on the final submission, because users can bypass an Ajaxified web application just as easily as a standard one, which means the final POST must be checked from top to bottom. So, this pattern gives us more powerful client-side usability but does not solve the underlying security problem at all.

We're going to modify the CRM application from the earlier chapters with our new Ajax patterns. For this validation example, we have to start by preparing the UI itself. Here is the original HTML for rendering the Customer Name and Address fields for input:

File 13

```
<tr>
  <th>Customer Name:</th>
  <td><input type="text" name="name"/></td>
</tr>
<tr>
  <th>Address:</th>
  <td><input type="text" name="address"/></td>
</tr>
```

It includes a label and an input field for each data value. It doesn't have any reasonable place to put an error message when validation fails. First, error messages should be conveniently colocated with the

Click Here to purchase this book now.

Validation Error Messages

In addition to displaying error messages next to the fields they are associated with, it is also common (and, dare we say, appropriate) to include a general message area that provides a summary of all error messages. Adding one is left as an exercise to the reader.

input fields they describe, so we'll add a new <*span*> element directly beside the input fields. The <*span*> tags will be marked with a specific CSS class so that we can control their look (in this case, we'll just style the text red). Plus, we'll update the input fields to each have a unique ID, which we can use to extract the values at runtime, and the new <*span*> tags also have IDs so we can fill them in with a new innerHTML after validation.

Second, we'll need to hook our validation code up to an event on the input fields. The standard event to hook for this purpose is the onblur event. This event fires whenever the user changes focus away from the field, whether by clicking elsewhere or tabbing away from it. We'll call a JavaScript method from the onblur event that will perform the validation. The method is called validateField(), and we'll examine it more in a minute. For now, know that the function takes four parameters:

- field id: The ID of the input field being validated
- required: Whether this field is a required field
- validation: The validation rule to execute on the data
- update: The ID of the field used to display the error message

The new version of the UI elements looks like this:

File 8

```
<tr>
  <th>Customer Name:</th>
  <td>
    <input type="text" id="name" name="name"
           onblur="validateField('name', 'required', 'name', 'nameError')"/>
  </td>
  <td colspan="2">
    <span style="border-bottom: solid 1px red;
                 color: red;" id="nameError">
    </span>
  </td>
</tr>
```

```
<tr>
  <th>Address:</th>
  <td>
    <input type="text" id="address" name="address"
           onblur="validateField('address', 'required', 'address', 'addressError')"/>
  </td>
  <td colspan="2">
    <span style="border-bottom: solid 1px red;
                 color: red;" id="addressError">
    </span>
  </td>
</tr>
```

Third, we need to write the method that calls the validation on the server. Its job is to launch an asynchronous request, passing in enough information to validate the field, and then update a named display element with the error message, if any. Our validateField() method first constructs a parameter list to append to the validation URL using the input parameters to the method. It then uses the Prototype library's Ajax.Updater to fire the request and fill in the display field with any error message generated.

File 8

```
function validateField(fieldname, required, validation, update) {
  var params = "type=" + validation    +
               "&required=" + required +
               "&value=" + $F(fieldname);

  new Ajax.Updater(update, validationUrl, {
    asynchronous: true,
    method: "get",
    parameters: params
  });
}
```

Finally, we need to create a server-based validation engine. You could call any standard platform validation engine you want: Struts validation, dyna-validation, Spring's Validator, the ASP.NET validation rules, a rules engine, whatever. Here, we've written a custom servlet that takes a field's value and the rules to invoke (required or not, plus specific rule) and returns either an empty string (meaning it succeeded) or an error message (for failure). Clearly, we'd add things such as i18n and SQL-injection protection if this were to be released to the public. The listing of that code, in its entirety, is on the next page.

Click Here to purchase this book now.

File 6

```
package ajaxian.book.crm.servlet;

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

public class ValidationServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        System.out.println(request);
        String required = request.getParameter("required");
        String type = request.getParameter("type");
        String value = request.getParameter("value");
        String message = "";
        if(required.equals("required")) message += validateRequired(value);
        out.println(message);
    }

    private String validateRequired(String input) {
        if (null==input || 0==input.length()) return "Field required";
        return "";
    }
}
```

When the user first sees the page, as shown in Figure 7.1, on the next page, it looks like any standard HTML form, waiting for input.

As the user tabs through the fields, leaving data that breaks the rules, the page updates without a refresh, giving the user instant feedback, as shown in Figure 7.2, on the following page.

Request Notification

The asynchronous server-side validation we just created works well. The user gets a pretty big benefit without too much of a cost. We do have one problem, though. The user is firing server-side events via a nonstandard mechanism. Rarely does a web application user expect the TAB key to establish a connection back to the server. Without that expectation, they might be surprised to find that bandwidth is consumed at this point and even more surprised when, a half second later, the UI suddenly pops up a block of red text next to the field they

Click Here to purchase this book now.

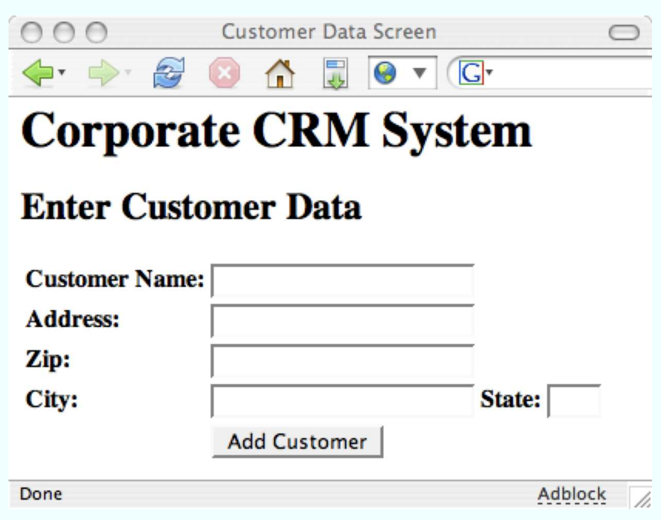


Figure 7.1: Form Waiting for Input

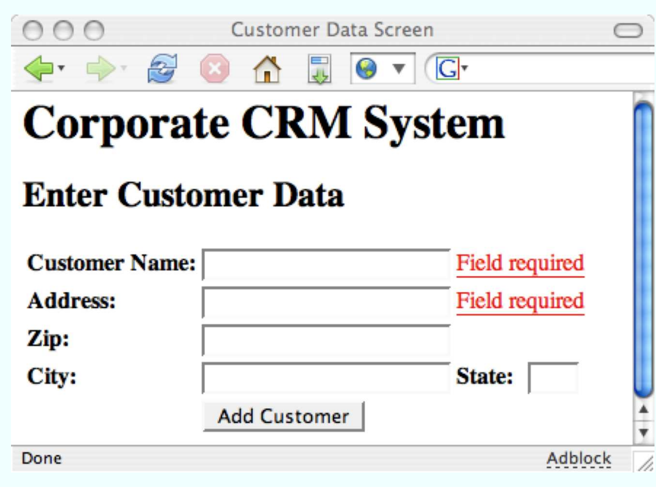


Figure 7.2: Form Displaying Validation Errors

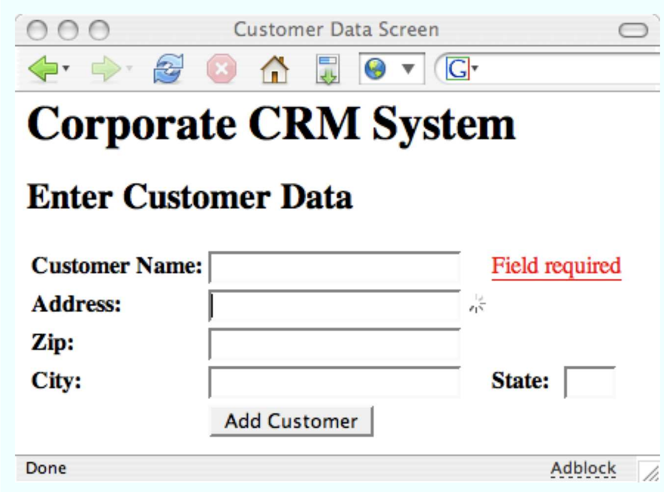


Figure 7.3: Form Processing Validation Request

just left. If you take into account the expected occurrence of network latency, suddenly you have the scenario of a user getting all the way to the bottom of a form before error messages start filling in at the top. How bad would it be if the error messages popped up in an area of the screen the user has already scrolled past? Fairly inconvenient, at the least.

The answer is to include a feedback mechanism that alerts the user that a request is in progress. Browsers typically accomplish this through a spinning/jumping/waving graphic in upper-right corner that animates only while a request is being processed. Ajax techniques can't take advantage of this UI convention, though, for two reasons: it is difficult to impossible to control the browser's request icon, and it can alert you to the status of only a single request at a time. With Ajax, and a technique like the validation described above, there can be multiple concurrent requests being processed.

The standard solution is to show an animated graphic that indicates a request in process. This is displayed inline, wherever the results of the request will be displayed, as shown in Figure 7.3 . If the graphic pops up immediately, the user knows right away that something is happening and where to look for the results. Multiple graphics can be shown simultaneously by embedding them in multiple containers

CLICK HERE to purchase this book now.

in the DOM. The current standard is to use an animated GIF image, which is quick to load and implies activity without having to actually poll the current status of the request.

First, we'll add some <*span*>s to the page to hold our progress indicators. In this case, the image is an animated GIF called progress.gif, which is just a spinning wheel. We'll add them between the input fields and the associated error message containers; this will place the notification GIF approximately where the error message will appear, so the eye is drawn to the appropriate place. We'll go ahead and make a hard link to the image, rather than loading it dynamically with JavaScript, though either would be acceptable. The browser will natively attempt to cache the image for the first container, and all subsequent containers will use the cached GIF, preventing needless round-trips to the server for the same file. We'll simply place the image in a <*span*> whose display: style is set to none. When we want to notify the user, we toggle the <*span*>. When the request is complete, we toggle it again. Here's the code:

File 7

```
<tr>
  <th>Customer Name:</th>
  <td>
    <input type="text" id="name" name="name"
           onblur="validateField('name', 'required', 'name', 'nameError')"/>
    <span id="nameProgress"
          style="display:none;"><img src="../progress.gif"/></span>
  </td>
  <td colspan="2">
    <span style="border-bottom: solid 1px red; color: red;"
          id="nameError">
    </span>
  </td>
</tr>
<tr>
  <th>Address:</th>
  <td>
    <input type="text" id="address" name="address"
           onblur="validateField('address', 'required',
                                 'address', 'addressError')"/>
    <span id="addressProgress"
          style="display:none;"><img src="../progress.gif"/>
    </span>
  </td>
  <td colspan="2">
    <span style="border-bottom: solid 1px red; color: red;"
          id="addressError">
    </span>
  </td>
</tr>
```

Click Here to purchase this book now.

Second, we have to update our request-generating code. In the previous example, we used the Prototype library's Ajax.Updater object to perform our round-trip. We'll extend that example here. The options collection contains four event hooks: onLoading,onLoaded,onInteractive, and onComplete, each corresponding to one of the four readystate values. Prototype simply implements the onreadystatechange hook and then publishes the specific events as those values arrive. We'll trap the onLoading and onComplete events, which allows us to show the image when the request begins and hide it once a response has been received. The values for the two events need to be function calls. Instead of simply calling Element.show() and Element.hide() directly, we'll wrap them in anonymous functions. If you don't do this, the onLoading call never completes, the validation result is never received, onComplete is never called, and the little spinning wheel becomes the only interesting thing about the page. Here's the code:

File 7

```
function validateField(fieldname, required, validation, update) {
   var params = "type=" + validation +
                "&required=" + required +
                "&value=" + $F(fieldname);

   new Ajax.Updater(update, validationUrl, {
      asynchronous: true,
      method: "get",
      parameters: params,
      onLoading:  function(request) {Element.show(fieldname + 'Progress');},
      onComplete: function(request) {Element.hide(fieldname + 'Progress');}
   });
}
```

Update Notification

Web surfers are largely trained to believe that something loaded on a page is static. They understand that in order to update the contents of a page, the page must be reloaded. The only cognitive exception to this rule is animations. The web-surfing population understands that certain graphics are not static but in fact loops of animation. These are expected to repeat the same set of information over time, though, and are not actually "dynamic" in any data-centric meaning of the term.

Ajax is all about breaking this particular expectation. That is, in fact, the core idea of Ajax: break free from the bonds of static information. But it goes against the foundation of most users' understanding of how the web works. This means we have to take special pains to ensure that when we do break this convention, users don't miss it.

Click Here to purchase this book now.

The primogenitor of this pattern is the famous Yellow Fade Technique, or YFT. Apparently created (or at least named) by the good folks at 37signals, the YFT is a simple trick. Simply choose a color (canonically and eponymously yellow), reset the background color of an element to this new color and then slowly transition it back to the original. The effect is to highlight an area of the page as though with a highlighter so as to draw the user's attention but to have that intrusive effect disappear so as not to detract from the overall look and feel of the page.

To do this, you could write some code that manipulates the background-color style of an element. In order to return the element to its original state at the end of the effect, you'll need to capture its original background-color. You'd have to deal with the fact that most browsers internally store colors in the form rgb(nnn, nnn, nnn). If you would prefer to work in hex notation (#789abc), then you would have to convert them yourself. Likewise, you would have to come up with some strategy for moving from the original value to the target value for each color (red, green, and blue) simultaneously to get a smooth transition.

Luckily, somebody else has already done that work for you. Previously, we've used the Prototype library to do server-side validation and progress notification. We're now going to layer the Script.aculo.us library on top of that to get the highlight effect.[1] We'll modify the sample application to use the YFT to alert you when the content of the City and State fields has been updated.

First, we don't have to change the HTML at all. We already have a container element with a unique ID that we can use for the highlight effect. It's the <*tr*> that holds the City and State fields. Its ID is rewrite.

File 9

```
<tr id="rewrite">
    <th>City:</th>
    <td>
      <input id="city" type="text" name="city"/>
    </td>
    <th>State:</th>
    <td>
      <input id="state" type="text" name="state"
             size='3' maxlength='2'/>
    </td>
</tr>
```

The second part is to update the getZipData() function to trigger the effect when the data has been loaded. Remember, XHR features the onreadys-

[1] http://script.aculo.us/

tatechange event to alert your code when the status of the request has changed. In this case, though, Prototype offers us another option. As we saw in Chapter 6, *Ajax UI, Part I*, on page 93, the Prototype library provides two new events, onSuccess and onFailure, so that we can write error-aware asynchronous methods. Our current version of getZipData() already uses onFailure to alert the user if the request fails:

File 13

```
function getZipData(zipCode) {
  new Ajax.Updater("rewrite", url, {
    asynchronous: true,
    method: "get",
    parameters: "zip=" + zipCode + "&type=html",
    onFailure: function(request) {
      assignError(request.responseText);
    }
  });
}
```

When the request fails, the assignError() function is called to display the message. We're now going to add a handler to the onSuccess method to perform the YFT. We use onSuccess instead of onComplete because onComplete will fire regardless of what's in the response. This would lead us to highlight City and State even if their data doesn't update. Instead, we use onSuccess, which fires only if the request returned data that ends up in the display fields:

File 23

```
function getZipData(zipCode) {
  new Ajax.Updater("rewrite", url, {
    asynchronous: true,
    method: "get",
    parameters: "zip=" + zipCode + "&type=html",
    onSuccess: function(request) {
       new Effect.Highlight('rewrite');
    },
    onFailure: function(request) {
            assignError(request.responseText);
    }
  });
}
```

The effect of this new handler is that the row containing City and State will go yellow whenever the request succeeds and then fade back to white over a one-second period. Bear in mind, as you learned in the previous chapter, you can affect the behavior of the transition by submitting options to the call. For example, you can change the transition to go from cornflower blue to white over three seconds with a linear transition by changing the call to the following:

CLICK HERE to purchase this book now.

```
new Effect.Highlight('rewrite',
                     { startcolor: '#92A4E2',
                       duration: 3.0,
                       transition: Effect.Transitions.linear } );
```

You can also choose the end transition color (endcolor) and the final color to use after the fade (restorecolor) if you need to.

Autocomplete

One of those things that often sets traditional thick clients apart from thin clients is the ability to quickly react to what the user is doing. For example, lots of locally installed applications can react to what a user is typing and make intelligent guesses about how to complete the word(s) for the user. Google (once again) showed that the same thing could be accomplished on the Web with Google Suggest. This feature has come to be known as *autocomplete*.

Script.aculo.us provides an amazingly simple-to-use version called the AutoCompleter. It watches an input field and sends a post parameter of the same name to a registered server endpoint. The results are rendered in another container node, allowing the user to choose from the results. The whole effect can be achieved with the addition of one container, one line of JavaScript, and a little simple CSS.

Let's add this feature to the sample CRM application. We'll prompt the user with potential Zip code matches based on what they are typing in the zip field. As they type into the zip field, we'll compare that against the list of available Zip codes and return those that are potential matches (the ones that start with the characters entered so far).

Let's start with a servlet that implements the autocompletion feature. Any reasonable production-quality version would use a database of Zip codes, and the SQL SELECT x WHERE zip LIKE 'y%' notation to retrieve values. To keep it simple for the book, the servlet will instead just keep an array of Zips as strings to compare against. Here's the servlet:

File 5

```
package ajaxian.book.crm.servlet;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletConfig;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Iterator;
import java.util.ArrayList;
```

CLICK HERE to purchase this book now.

```
public class AutoCompleteServlet extends HttpServlet {
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws IOException {
              System.out.println(request);
         String[] zips = new String[] {
            "10010", "11035", "27707", "31000", "32230", "34434",
            "45555", "46666", "46785", "46699", "49999", "53711", "53703" };

         ArrayList results = new ArrayList();

         String val = request.getParameter("zip");
         for(int i=0;i<zips.length;i++) {
            if(zips[i].startsWith(val)) results.add(zips[i]);
         }

         String message = "<ul>";
         Iterator iter = results.iterator();
         while(iter.hasNext()) {
            message += "<li>" + (String)iter.next() + "</li>";
         }
         message += "</ul>";


         response.setContentType("text/html");
              PrintWriter out = response.getWriter();

        out.println(message);
    }
}
```

Next, we'll have to add the Ajax.AutoCompleter and a container <*div*> to hold the responses we get from the server. The entire update to the UI is as follows:

File 22

```
<tr>
  <th>Zip:</th>
  <td>
    <input autocomplete="off" onblur="getZipData(this.value)"
           type="text" name="zip" id="zip"/>
    <div class="auto_complete" id="zip_values"></div>
  </td>
  <script type="text/javascript">
    new Ajax.Autocompleter('zip', 'zip_values',
                           '/ajaxian-book-crm/autoComplete', {})
  </script>
  <td id="zipError" style="color: red"></td>
</tr>
```

CLICK HERE to purchase this book now.

First, we had to make a minor change to the zip input field itself. We added the autocomplete="off" attribute, which prevents the browser from attempting to fill in the value itself. This would preempt our JavaScript version and nullify the whole exercise, so we'll disable it. Next, we have to add a container to hold the results; that's the <*div*> named zip_values. Finally, we add a <*script*> block to invoke the Ajax.AutoCompleter. The first parameter is the id of the input field to be autocompleted, the second is the ID of the container to display the results, the third is the server endpoint to send the request to, and the final parameter is a collection of options.

In our case, we're not using any of the optional parameters since the defaults work just fine for this purpose. However, the options you have to customize the behavior of the AutoCompleter are:

- paramName: A name to use for the value sent to the server. This defaults to the name of the target input field.
- frequency: How often to check for changes to the input field and send the request (defaults to 0.4 seconds)
- minChars: How many characters the user has to enter before the first request is sent (defaults to 1)
- afterUpdateElement: A hook invoked after the values are returned and set into the target container

Script.aculo.us also provides another object, AutoCompleter.Local, which uses a locally cached list of values instead of making round-trips to the server. This would increase speed at the expense of stale data.

To finish the example, we just have to make the results look pretty. Without any style help, the results will be displayed in a transparent <*div*> as a series of bulleted list items, without keyboard navigation. Clicking on one with the mouse would be the only way to select an entry from the list. We are using the styles provided by Script.aculo.us to make our list entries navigable and pretty, as shown here:

File 22

```
<style>
  div.auto_complete {
    width: 350px;
    background: #fff;
  }
  div.auto_complete ul {
    border:1px solid #888;
    margin:0;
    padding:0;
```

CLICK HERE to purchase this book now.

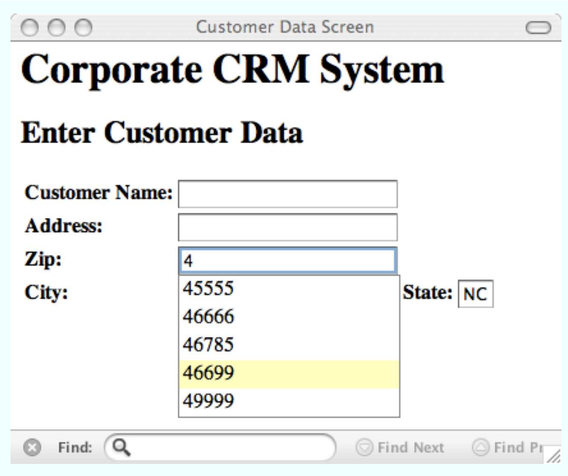


Figure 7.4: Autocomplete in Action

```
    width:100%;
    list-style-type:none;
  }
  div.auto_complete ul li {
    margin:0;
    padding:3px;
  }
  div.auto_complete ul li.selected {
    background-color: #ffb;
  }
  div.auto_complete ul strong.highlight {
    color: #800;
    margin:0;
    padding:0;
  }
</style>
```

Figure 7.4 shows the final result. Notice how the effect is like a drop-down box. The <*div*> has a narrow black border, the individual items are displayed without list bullets, and as you key up and down the list, the items highlight with (in this case) a pale yellow. Pressing enter while an item is highlighted, or clicking one with the mouse, causes that value to be set into the input field.

CLICK HERE to purchase this book now.

7.2 It Isn't All Just Wine and Roses...

Ajax is fantastic. It opens the Web to a whole new way of developing and delivering applications to your users. Largely, it changes the experience of using a web app from *reading* to *using*. As long-time instructors and trainers, we know firsthand the value of interaction in keeping students engaged and happy. The same phenomenon applies to applications. If your application is passive and makes your users passive consumers, then the application will not capture your users' attention. An interactive version, however, has the power to excite.

Even though Ajax has this power to change the Web so radically, it behooves us all as developers to remember why the Web enjoys such broad acceptance. It is based around certain standards (technical and visual) that have allowed users of all stripes to take advantage of services provided there. Those standards, some written and some simply understood, are vital to the success of all applications on the Web, whether or not they use Ajax.

The key to successful Ajaxification is to not ignore important conventions. Certain laws of the land made the Web so popular and accessible, in ways that other applications and technologies never were. As you add this new technology into your application, think both tactically and strategically. Ask yourself the following questions:

- "Is what I'm adding increasing the usability of my application, or the length of my resume?"
- "Does it break an ingrained habit of my users?"
- "Is the value worth the cognitive dissonance such a break will cause for my users?"

Tactically, the change might increase the usability of this single page but strategically reduce the usability of the application as a whole.

We'll walk through some of the biggest antipatterns to watch out for. This list is not exhaustive. When in doubt about something you are working on, check it against our previous smell-test questions. And keep in mind that the key is usability and fun: if it increases both, then do it!

Watch That Back Button!

Two features set the World Wide Web apart from everything that came before it: the back button and the bookmark. Applications histori-

Click Here to purchase this book now.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help programmers stay on top of their game.

Visit Us Online

Pragmatic Ajax
`pragmaticprogrammer.com/titles/ajax`
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

Join the Community
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/ajax.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com