Prototype & Scriptaculous IN ACTION

SAMPLE CHAPTER

Dave Crane
Bear Bibeault
with Tom Locke

Foreword by Thomas Fuchs

MANNING

Prototype and Scriptaculous in Action

by Dave Crane
Bear Bibeault
with Tom Locke

Chapter 4

Copyright 2007 Manning Publications

brief contents

PART IV ADVANCED TOPICS 357

4 Using Prototype's Advanced Ajax Features

In this chapter

- Prototype's advanced Ajax classes
- Using HTML and Ajax
- Comparing data- and content-centric Ajax
- Measuring HTTP traffic

This chapter will conclude our examination of the different styles of Ajax, of what Ajax can bring to a web application, and how Prototype makes Ajax development easy. In chapter 2, we introduced the QuickGallery application, a non-Ajax web app that we were going to convert to use Ajax, in order to eliminate the full-page refresh associated with every request made to the server and change the stop-start pattern of user interaction. In chapter 3, we developed two Ajax-powered versions of the QuickGallery application, using XML and JSON. Both transmitted updates from the server as structured data, with the client-side JavaScript containing all the logic for parsing this data and generating updates to the user interface. In terms of the types of Ajax that we identified in section 3.1.1, the implementations that we presented in chapter 3 clearly fitted the content-centric model.

In this chapter, we're going to rework QuickGallery, still using the content-centric approach to Ajax, to see if we can cut down on the amount of client-side code we have to write. That is, the server will generate updates as fragments of HTML directly, relieving the client-side code of the burden of parsing data and generating HTML markup in one fell swoop. All we have to do is read the response data and stitch it into the document using the `innerHTML` property.

In fact, we don't even need to do that. In section 3.1.2, we alluded to "deluxe models" of the Ajax helper class in the Prototype library. As we will see, Prototype provides us with a special helper class, the Ajax.Updater, that will make working with Ajax even easier. We'll begin this chapter, then, by looking at the Ajax.Updater and related classes. We'll then move on to develop an implementation of QuickGallery that makes use of these classes, and conclude by evaluating and comparing the various styles of Ajax that we've encountered in chapters 3 and 4.

4.1 Prototype's advanced Ajax classes

In the previous chapter, we looked at the Ajax.Request class supplied by Prototype.js. Ajax.Request provides an easy-to-use wrapper around the XMLHttpRequest object, but it still leaves the developer with the job of writing code to make sense of the response. Prototype.js also provides us with a pair of more advanced Ajax helper classes, specifically designed to support content-centric Ajax. In this section, we'll look at these classes in more detail before making use of them in our QuickGallery application.

4.1.1 Ajax.Updater

Ajax.Updater extends Ajax.Request in a very convenient way. When using the content-centric approach to Ajax that we described earlier, we will generally

want to read the server response as a string, and then apply it as the `innerHTML` property of a DOM element somewhere on the page. Ajax.Updater saves us the trouble of doing this manually every time. When we create the object, we specify a URL as an argument, as before, but also a container element to be populated by the response. The container argument may be either a single DOM element, or a pair of DOM elements, one of which will be populated if the response is successful, the other if it is unsuccessful.

Because Ajax.Updater extends Ajax.Request, we can still access all the configuration options of the parent class when using the Ajax.Updater, as outlined in table 3.1. In addition to these, some additional configuration options are provided, as listed in table 4.1.

Table 4.1 Optional properties of the Ajax.Updater object

Name	Default value	Comments
`insertion`	`null`	A Prototype.js Insertion object (see chapter 11) that, if present, will be used to insert the response content into the existing markup for the target element. If omitted, the response replaces any existing markup.
`evalScripts`	`false`	When a DOM element's content is rewritten using innerHTML, `<script>` tags present in the markup are ignored. The Ajax.Updater will strip out any `<script>` tags in the response. If `evalScripts` is set to `true`, the content of these scripts will be evaluated.
`onComplete`	`null`	Supplied by Ajax.Request-see table 3.1. Programmatic callbacks can still be used with the Ajax.Updater. These will be executed after the target's content has been updated.

Ajax.Updater neatly encapsulates a commonly used way of working with Ajax. We'll make use of this object in our QuickGallery application too, and look at its advantages and disadvantages compared with direct use of the Ajax.Request.

4.1.2 Ajax.PeriodicalUpdater

The Ajax.PeriodicalUpdater helper class adds a final twist to the Prototype Ajax classes, once again automating a commonly used Ajax coding idiom. Ajax.PeriodicalUpdater extends Ajax.Updater by managing a timer object, so that the request for fresh content is made automatically with a given frequency. Using this object, an automatically updating stock ticker or news feed, for example, can be created with a minimum of fuss. The Ajax.PeriodicalUpdater class is a direct descendant of Ajax. Updater, and, as such, has access to all of its configuration options. New options provided by Ajax.PeriodicalUpdater are shown in table 4.2.

Table 4.2 Optional properties of the Ajax.Updater object

Name	Default value	Comments
`frequency`	`2`	Frequency of automatic update of the content, measured in seconds.
`decay`	`1`	Decay in frequency of updates when the received response is the same as the previous response. Set to a value less than 1 to increase the frequency on successive polls, or greater than 1 to decrease the frequency (i.e., increase the length of time between updates).

The Ajax.PeriodicalUpdater class also introduces two useful methods, `stop()` and `start()`, which will pause and resume the automated polling of the server.

Ajax.PeriodicalUpdater is useful for situations where regular updates from the server are required. It should be noted, though, that it makes it extremely easy to increase both the amount of HTTP traffic that an application generates, and the load on the server. It should therefore be used with caution.

As you can see, Prototype provides us with some very useful tools for content-centric Ajax. In the next section, we'll put them to use in our QuickGallery application. First, though, we'll round off our tour of the Ajax helper classes provided by Prototype with a quick mention of a recently added feature.

4.1.3 Ajax.Responders

In version 1.5 of Prototype, a new Ajax feature was added to the library. We've already seen in chapter 3 how to attach callback handler functions to an Ajax.Request object by specifying an `onComplete`, `onSuccess`, or `onFailure` option in the options passed to the constructor. These functions will be triggered only when a specific response comes in. In most cases, this is exactly what we need, but in a few cases, we might also want to be notified whenever any Ajax response comes in.

The Ajax.Responders object looks after this requirement for us. It maintains a list of objects that will automatically be notified whenever any Ajax request is made. We'll see the Ajax.Responders object in action later in this chapter.

That's enough of an introduction for now. In the next section, we'll return to the QuickGallery example and see how these extra Ajax helper classes operate.

4.2 Using HTML and Ajax

In this section, we'll develop the third Ajax-based version of QuickGallery. We originally applied Ajax to QuickGallery in order to get rid of unnecessary full-page refreshes in the app, and both of the implementations in chapter 3 succeeded on that score. However, we had to develop a lot of JavaScript code to handle the response data and manually update the user interface. On first glance,

Ajax.Updater looks like it will make things much simpler for us on the client. All we need to do is tell it which DOM node we want to update, and generate the request as we did previously.

In this section, we'll put those first impressions to the test and see how much added convenience Ajax.Updater really offers us.

4.2.1 Generating the HTML fragment

Let's get started, then. The most complex part of the user interface in QuickGallery is the thumbnail images, so we'll begin by generating HTML fragments for that. Because our server-side code is well-factored, we don't need to even touch our business logic, but simply alter the template that generates the response. Listing 2.2 presented the original template for the pre-Ajax application, and listing 3.2 the modified template for our XML-powered version of the app. Listing 4.1 shows how we've modified the template to generate a fragment of HTML.

Listing 4.1 contentUpdate/images.php

```
<?php
require('../config.php');
require('images.inc.php');
if (count($imgs)>0){
  foreach ($imgs as $i => $value){
    $full_img=implode('/',array($path,$value));
?>
<div class='img_tile'>
  <img border='0'
     src="<?php echo $img_pre_path.$full_img ?>.thumb.jpg"
     onclick="showCloseup('<?php echo $img_pre_path.$full_img ?>.jpg')"/>
  <br/>
  <?php echo $value ?>
  </a>
</div>
<?php
  }
}
?>
```

❶ **Import business logic**

❷ **Render image tile**

The template is very straightforward. We simply import the business logic code ❶ that generates the data on thumbnails, subfolders, etc., for the current folder, and then iterate over the list of images, outputting a bit of HTML for each one ❷. So far, so good. Now let's take a look at the client.

4.2.2 Modifying the client-side code

On the client side, our task is equally simple. In the `load()` function, we create an Ajax.Updater object rather than an Ajax.Request, and pass it a reference to the DOM element that we want to receive the content. The code required to do this is as follows:

```
function load(newPath){
  if (newPath!=null){ currPath=newPath; }
  new Ajax.Updater(
    "images",
    "images.php?path="+currPath,
    {
      method: "get",
      onComplete: function(){
        Element.hide(ui.closeup);
      }
    }
  );
}
```

Creating the Ajax.Updater looks pretty familiar after our work with Ajax.Request, but there is an extra argument present in our call to the constructor. Let's stop and look at the arguments we passed into Ajax.Updater. The first is the name of the DOM element, in this case images. We've passed in a string here, but Ajax.Updater will also accept a reference to the DOM element itself. Most Prototype functions and objects that work with DOM nodes provide this flexibility, because the `$()` function makes it so simple to resolve either as a programmatic reference to the element itself.

The second argument is the URL to our server-side resource, and the third argument is the collection of options. Ajax.Updater inherits all of the functionality of the Ajax.Request class, so it understands all of the options that Ajax.Request does (see table 3.1), and it operates on the same defaults. It also understands a few more options of its own, as we saw in table 4.1. For now, all we need to do is pass in the HTTP method that we're going to use, and a small function that we'll execute when the request completes, to ensure that the close-up DOM element is hidden from view (otherwise we might not be able to see our refreshed thumbnail view, as the two share the same portion of the screen).

So, Ajax.Updater has made life a lot easier for us. The server-side code is no more complex than before, and the client-side code is markedly simpler. However, in the case of our application, there is a catch. We'll look at the problem—and solutions—in the next section.

4.2.3 Updating multiple DOM elements

The code we've presented so far is admirably simple, but we have a problem. When we navigate to a new folder, we need to update the breadcrumb trail, the subfolders list, and the thumbnails. We've laid these out as three separate DOM elements on the page, but so far we've only updated one of them. The limitation of Ajax.Updater without evaluating scripts in the response is that the class updates only one element in an Ajax request.

Outlining the problem

So what are our options? We could create three Ajax.Updater objects, one for each element, but this would be extremely inefficient in several ways.

First, we'd be generating three HTTP requests. HTTP requests contain considerable bandwidth overhead in terms of the headers in the request and response, so we'd be adding to the bandwidth use of our app (see appendix A for more details of the HTTP protocol and appendix B for techniques for measuring HTTP traffic).

Second, on the server side, we'd need to execute our business logic three times, once for each request. In our case, that's three hits to the filesystem, and in other applications it might translate to three hits to the database, to some other network resource, or three runs of an expensive calculation. Either way, we're increasing the server load significantly. We could do the calculations once and store the results in session, but this would require us to write some tricky synchronization logic to ensure that the session gets tidied up at the right time. Remember, we're going down this route to make our client-side coding simpler. We don't want to simply trade it for more complex server-side code.

Finally, we'd need to account for the fact that the network is unpredictable and unreliable. We don't know in what order our requests will be processed or the responses will be returned. We don't want to update each element as the response returns, because it leaves the user interface in an inconsistent state. When we consider that we run the risk of some requests failing while others succeed, we face the problem of this inconsistency persisting indefinitely.

So, we've persuaded ourselves that we need to update all elements of the user interface in a single request. We could regenerate the entire page as a single top-level DOM element, but that would take us back to a full-page refresh. Our UI is pretty sparse at the moment, but if we had dressed it up a bit more, we'd be back in the world of flickering pages and clunky stop-start interactions, only with more code to maintain! Another dead end.

Fortunately, there is more than one way out for us that conveniently allows us to introduce some of Prototype's more advanced Ajax features. We'll see how it's done in the next section.

Attaching scripts to the response

Our first solution continues to use the Ajax.Updater object. Ajax.Updater allows us to attach script content to the response. Both the subfolder list and the breadcrumb trail are very simple in terms of the HTML behind their user interfaces, and if we're willing to put up with generating those interfaces in the JavaScript, we can pass the necessary data up with our request.

Here's how it works. When we add markup to the DOM using `innerHTML`, any `<script>` tags in the HTML text will be ignored by the browser. However, Ajax.Updater has a mechanism that allows it to extract the content of these script tags and evaluate them immediately after updating the DOM element. We can use this to generate calls to update the breadcrumbs and subfolders list, and achieve our aim of updating all user interface elements with a single request. Let's see what we need to do to make it work.

Our first job is to switch the feature on when we create the Ajax.Updater object. This is accomplished simply by passing in an extra option to the constructor, as follows (changes from the previous versions of this code, presented in section 3.2.3, are in bold):

```
function load(newPath){
  if (newPath!=null){ currPath=newPath; }
  new Ajax.Updater(
    "images",
    "images.php?path="+currPath,
    {
      method: "GET",
      evalScripts:true,
      onComplete: function(){
        Element.hide(ui.closeup);
      }
    }
  );
}
```

The `evalScripts` option simply tells the updater to execute any scripts that it extracts from the response.

Now that we've told it to do that, we need to generate the scripts. Listing 4.2 shows the modified PHP template, which corresponds to images.php in the contentScript directory.

Listing 4.2 images.php with added script tags

```
<?php
require('../config.php');
require('images.inc.php');
$folder_list="";      ❶ Compute folder list
if (count($subdirs)>0){
 $folder_list='"'.implode('","',$subdirs).'"';
}
?>
<script type='text/javascript'>  ❷ Generate script tag
  showBreadcrumbs();
  showFolders([<?php echo $folder_list ?>]);
  imgCount=<?php echo count($imgs)?>;
  if (imgCount>0){
    Element.show(ui.images);
  }else{
    Element.hide(ui.images);
  }
</script>

<?php
if (count($imgs)>0){
  foreach ($imgs as $i => $value){
    $full_img=implode('/',array($path,$value));
?>
<div class='img_tile'>
  <img border='0'
     src="<?php echo $img_pre_path.$full_img ?>.thumb.jpg"
     onclick="showCloseup('<?php echo $img_pre_path.$full_img ?>.jpg')"/>
  <br/>
  <?php echo $value ?>
  </a>
</div>
<?php
  }
}
?>
```

The script that we've generated ❷ simply collates the list of subfolders as a string ❶ and calls two JavaScript functions that we've defined statically. The `showBreadcrumbs()` function needs no arguments because the client-side code already knows the destination folder's path, having passed it down in the request. The second function, `showFolders()`, takes a JavaScript array as an argument, which we populate with the list of subfolders that we generated earlier. The line

```
showFolders([<?php echo $folder_list ?>]);
```

will generate code looking like this:

```
showFolders(["trees","foliage","flowers"]);
```

Or, if no subfolders are present, it will simply look like this:

```
showFolders([]);
```

We need to support the generated script by providing the functions that it calls in our static JavaScript. Fortunately, we've already written them when we developed the XML-based version of our Ajax app, so we need only repeat that here. The `showBreadcrumbs()` method can be reused unaltered from listing 3.2. The `showFolders()` method needs a little bit of tweaking, as shown here (changes in bold, again):

```
function showFolders(folders){
  if (folders.length==0){
    Element.hide(ui.folders);
  }else{
    var links=folders.collect(
      function(value,index){
        var path=[data.path,value].join("/");
        return "<div onclick='load(\""+path+"\")'>"+value+"</div>";
      }
    );
    Element.show(ui.folders);
    ui.folders.innerHTML=links.join("");
  }
}
```

In the data-centric approach, we assigned the global value `data.folders` when we parsed the response, and iterated over that. Here, we're simply using the locally scoped variable passed in as an argument.

The astute reader will have noticed at this point that we've slipped from a purely content-centric approach to a mixture of content-centric and script-centric. That is, we're generating a mixture of HTML markup and client-side code. We noted in our earlier discussion of script-centric Ajax, in section 3.1.1, that this approach presents a danger of introducing tight coupling between the client- and server-side code bases. We also noted that the best way to avoid this was to define a high-level API in the static client code, and simply call that API in the generated code. This is what we've done here.

In addition to reducing coupling, keeping generated code to a minimum makes the application easier to maintain. Static JavaScript is easier to debug than dynamically generated code, and it is also more amenable to testing. We have also reduced the size of the response by abstracting out the common logic into a static API that needs be downloaded only once.

We've now implemented the complete QuickGallery app in a mostly content-centric way, with a bit of script-centric Ajax thrown in. The Ajax.Updater class looked at first like it was going to eliminate most of our code, and in a simpler application it might have done just that. Our requirement to simultaneously update more than one DOM element made us dig into some of the more advanced features of the Ajax.Updater object, but, even so, we've managed to simplify our client-side code base considerably.

Happily, Ajax.Updater has shown that it is capable of addressing the problem of updating multiple elements from a single response. Only one element can be updated as pure content, but we can pass additional instructions in the response as JavaScript.

Before we leave this topic of multiple-element refreshes, though, we should note that there's a second approach that we can take to solving this problem, using a different set of features from Prototype's Ajax support classes. We'll take a look at that in the next section.

Responding to custom headers

As an alternative to adding script tags to the response body, we can encode the additional information in the HTTP headers of the response. Recent builds of Prototype have added support for this, using the compact JSON syntax that we saw in the previous chapter. There are two parts to this approach, so let's take each in turn.

First, if a response contains a header called X-JSON, Prototype's Ajax classes will try to parse it and pass it to the callback functions as an extra parameter. In order to generate this header, we need to modify our PHP script, contentJSON/images.php, as follows (changes shown in bold, again):

```
<?php
require('../config.php');
require('images.inc.php');
$folder_list="";
if (count($subdirs)>0){
 $folder_list='"'.implode('","',$subdirs).'"';
}
$json='{ folders:['.$folder_list.'], count:'.count($imgs).'}';
header('X-JSON: '.$json);
?>
<?php
if (count($imgs)>0){
  foreach ($imgs as $i => $value){
    $full_img=implode('/',array($path,$value));
?>
```

```
<div class='img_tile'>
  <img border='0'
     src="<?php echo $img_pre_path.$full_img ?>.thumb.jpg"
     onclick="showCloseup('<?php echo $img_pre_path.$full_img ?>.jpg')"/>
  <br/>
  <?php echo $value ?>
  </a>
</div>
<?php
  }
}
?>
```

When the response comes back, the body will now contain only the HTML for the main panel, and an additional header looking something like this:

```
X-JSON: { folders: ["trees","foliage"], count: 6 }
```

In this case, we're indicating that the current folder contains six images and has two subfolders, called "trees" and "foliage".

The second part of the solution involves picking this header up on the client and unpacking the data. Prototype will handle the evaluation of the JSON expression for us—the first thing we'll see of it is the parsed object appearing as an argument to our callback function. We could parse the JSON object within our main callback handler, but instead we're going to define a separate responder to handle it, using the Ajax.Responders object. This will give us the option of updating the subfolders list whenever we make an Ajax call, and not only when we're changing directory.

To set this up, we need to register a responder. The Ajax.Responders object provides a `register()` method for us, to which we pass our responder object. The responder can define any of the callbacks available to the Ajax.Request object. Here, we'll simply provide an `onComplete()` method. Let's look at the code now.

```
Ajax.Responders.register(
  {
    onComplete:function(request,transport,json){
      showBreadcrumbs();
      showFolders(json.folders);
      if (json.count!=null){
        if (json.count>0){
          Element.show(ui.images);
        }else{
          Element.hide(ui.images);
        }
      }
    }
  }
);
```

The `onComplete()` callback takes three arguments. The first is the Ajax.Request object that has received the response, the second is the underlying XHR transport, and the third is the parsed X-JSON header. This object contains all the information we need, so we can then call our existing API to update the breadcrumb trail and the folders list as before.

The Ajax.Updater object, then, is capable of combining ease of use with flexibility in handling multiple elements, and it can do so in more than one way. We've now completed implementations of QuickGallery using a variety of different Ajax techniques. We'll compare these in section 4.3, but first we're going to look at the final Ajax helper object that Prototype provides.

4.2.4 Automatically updating content

We've now solved the issue of updating multiple elements within a content-based approach in two ways, by using `<script>` tags and JSON-formatted headers. Along the way, we've seen practical use of two of the three advanced Ajax helpers that we introduced in section 4.1. Before we move on to compare content- and data-centric Ajax, we'll briefly take a look at the third of the advanced Ajax helpers, the PeriodicalUpdater.

There are a number of use cases in which it is desirable for the server to be able to notify the browser of updates. HTTP is not built to support this model of interaction—all interactions must be initiated by the browser. A common workaround is for the browser to poll the server at regular intervals for updates. (This is not the only way of implementing a push of data from server to client, but that's outside the scope of our discussion here.)

Let's suppose that we want the images in the current folder to automatically update at regular intervals, so that we can see new images posted to the site. Using plain JavaScript, we'd need to start creating timer objects using `setTimeout()`, but Prototype wraps all this up for us in the Ajax.PeriodicalUpdater object.

To make the QuickGallery poll the server for updates, we only need to alter a couple of lines of code. Using Ajax.Updater, our `load()` method read as follows:

```
function load(newPath){
  if (newPath!=null){ currPath=newPath; }
  new Ajax.Updater(
    "images",
    "images.php?path="+currPath,
    {
      method: "get",
      evalScripts: true,
```

```
        onComplete: function(){
          Element.hide(ui.closeup);
        }
      }
    );
  }
```

To make our Updater poll the server, we simply need to replace Ajax.Updater with Ajax.PeriodicalUpdater, as shown here:

```
function load(newPath){
  if (newPath!=null){ currPath=newPath; }
  if (updater){
    updater.stop();
  }
  updater=new Ajax.PeriodicalUpdater(
    "images",
    "images.php?path="+currPath,
    {
      method: "get",
      evalScripts: true,
      frequency: 10,
      onComplete: function(){
        Element.hide(ui.closeup);
      }
    }
  );
}
```

We also added a frequency value in the options object. This specifies the time between receiving a response and sending out the next request, in seconds. Here, we've set a ten-second delay. Tuning this parameter is very application-specific, and it boils down to a trade-off between responsiveness and server load.

That's all there is to the Ajax.PeriodicalUpdater object. We don't have a desperate need for automatic updates in our gallery app, so we won't be carrying this change forward as we develop the app further, but hopefully we've demonstrated how easy it is to add that functionality if needed. We'll get back on track now, and return to the debate between content- and data-centric Ajax.

We've now implemented no less than four Ajax-based versions of the QuickGallery application that we described in chapter 2, each of which reproduces the functionality of the original application completely. In chapter 3, we developed two data-centric versions, in which the client-side code parsed raw data sent by the server, in the form of XML and JSON. In this chapter, we developed two content-centric implementations of the app, in which the server updated the main panel by directly generating the HTML, and updated secondary elements by adding extra `<script>` tags, or by passing JSON data in the header.

Before we go on to add any new functionality, which we'll do in chapter 12, we have a decision to make: we must decide whether to follow the data-centric or content-centric approach. We'll compare the two approaches in the following section, with an eye to seeing which will make life easiest for us as we begin to add new features.

4.3 Comparing data- and content-centric Ajax

We've implemented two versions of the application using Ajax, each with two variations, and now we face a difficult choice. We can easily draw up a long wish-list of new features for the QuickGallery application, and can envisage several additional months of development work implementing them all. In order for this development to be effective, we need to opt for either a data-centric or a content-centric approach. How are we going to make this decision?

There are several criteria that we can take into consideration, such as ease of development, support for the approach by our toolset, the efficiency and performance of the application, and how future-proof each solution is as our requirements expand. Breaking down our assessment in this way won't get us off the hook entirely—we'll still have a difficult decision to make at the end of the day, but at least it will be an informed decision. So let's consider each of our criteria in turn.

4.3.1 Considering ease of development

Ease of development cannot be measured in a hard and fast way, as it is ultimately subjective. Let's begin with the assumption that all code is difficult to write, and therefore the less code written, the easier the project is. It isn't as simple as that in reality, of course, and we'll unpack some of the nuances shortly, but this approach allows us to put forward some numbers to start the discussion.

Table 4.3 lists the total size of the files of each type in our three solutions in bytes, as reported by the Unix `ls` command. Numbers in parentheses indicate files reused without modification by an Ajax project from the non-Ajax project. The total for JavaScript files excludes the size of the Prototype libraries, as it took us negligible effort to download them and start using them.

The first thing we can see from these numbers is that all three Ajax projects required more code to be written than their non-Ajax counterpart. While the server-side code became simpler, we added a lot of client-side JavaScript. Of the two Ajax projects, the data-centric one required almost twice as much code as the content-centric one.

Table 4.3 Size of the QuickGallery projects by file type (in bytes)

Solution	PHP	HTML	JavaScript	Total
non-Ajax	4175	0	0	4175
content-centric Ajax	3841 (3122)	421	1659	5936
data-centric Ajax (XML)	3599 (3122)	433	3218	7250
data-centric Ajax (JSON)	3451 (3122)	421	2738	6610

As we already noted, not all code is equally difficult to write, either. Simply by virtue of using two programming languages instead of one, we've presumably ramped up the difficulty level by introducing Ajax. We haven't had to write very much additional PHP, and what we did write was simple template stuff, but looking forward we still have to maintain our existing business logic code. So the main extra burden comes from the JavaScript.

But we knew that already. Our immediate concern is the difference between content-centric and data-centric Ajax. The content-centric Ajax required less code. When we look at the extra code required by the data-centric approach, we see a lot of involved user interface generation, and unpleasant DOM manipulation routines in the XML case too. If we accept that not all JavaScript code is equally difficult to write, the content-centric approach is even more of a clear winner here.

Let's note another point in the content-centric solution's favor. If we compare the two PHP templates that we had to write, the one for the content-centric solution is largely a direct cut and paste of PHP from the non-Ajax code. It isn't elegant reuse, but it is easier to write than the XML- or JSON-based templates, both of which required us to figure out a new data format.

Is the content-centric solution always the clear winner, then? There is one final point that we ought to consider. Our PHP template for the content-centric approach benefited from the fact that our legacy app was HTML-based. In another situation, we might have inherited an XML document format, in which case the data-centric Ajax solution might require very little work on the server. Reuse is king as far as ease of development is concerned, and reuse is highly context-dependent. If we're "Ajaxifying" a straightforward HTML-based web application, we'll find more scope for reuse in the content-centric option. In an enterprise setting, the XML data-centric solution might hold its own.

However, all in all, the content-centric approach seems to have won this round. Let's move on to our next criterion.

4.3.2 Fitting the tools

Ease of development in itself is a good thing. When considering which style of programming to use in a project, though, it's also useful to look at any supporting libraries that we're making use of, and ask ourselves which styles they favor. This ties in closely to ease of use, but it also gives an indication of how the library might develop in the future. If we're having to fight against the library to achieve our ends, and work around the recommended usage of that library or use undocumented features, future versions of the library might break those workarounds. Being left to rely on an old, unsupported version of a library is not a comfortable situation.

The main library we've used so far is Prototype.js. Prototype is very much geared toward doing things in a content-centric way. The specialist Ajax helper classes that we saw in this chapter are all geared toward the content-centric approach. Further, so is the entire Ruby on Rails development movement, both in the design of the framework and in the opinions expressed by leading Rails developers. We don't need to be using Rails for this to be a consideration. Prototype's main author, Sam Stephenson, is a core Rails committer, and its a fair bet that Prototype will continue to evolve to support the content-centric way of doing Ajax. Recent developments in Prototype (and in Ruby on Rails) are exploring script- and data-centric approaches, but only as complements to the core content-centric approach.

So, round two goes to the content-centric approach too. The next criterion that we put up for consideration was the performance of the app, so let's look at that.

4.3.3 Comparing performance

While it is important to work in a style that makes development easy, it's also important that our methodology produces code that runs efficiently. After all, we'd rather pay the price of difficult development once than the price of poor performance continually.

There has been a lot of debate about whether Ajax increases or decreases the efficiency of an application. Tuning in to this debate, we've heard good things about the reduction in traffic that comes from not continually sending boilerplate markup over the wire, and concerns expressed about the increased network traffic resulting from too many little messages being exchanged between the client and the server. Being cautious types, we aren't going to believe either argument until we've seen some hard numbers, and fortunately, there are a number of tools out there that can help us get those numbers.

We've defined a simple methodology for analyzing live HTTP traffic from any web application running in any web browser. We describe all the technical details

in appendix B, but as this is the first time we've used our analysis toolkit, let's run through the procedure right now, before we look at the numbers.

The bandwidth that an application consumes is unlikely to be regular. All the time that the user is interacting with the app, there will be traffic between the browser and the server. Because we're interested in the overall impact on the network of our app, we're going to have to define a test script to represent a typical session with the server. Our test script for working with the QuickGallery application is quite straightforward and is outlined in table 4.4. In order to write the test script, we created a sample set of images to be viewed by the QuickGallery, taken from Dave's copious collection of photos that sit on his computer doing nothing. (I knew they'd come in useful one day!)

Table 4.4 Script used for monitoring the performance of QuickGallery

Step	Description
1	Browser starts on the home page
2	User navigates into the "animals" folder, which contains 8 images
3	User clicks on the first thumbnail to view close-up
4	User returns to the home folder by the breadcrumb trail
5	User navigates to the "plants" folder, which contains 38 images
6	User navigates into the "foliage" subfolder (only 2 images)
7	User returns to the "plants" folder
8	User navigates into the "trees" subfolder, which holds 7 images
9	User clicks the first thumbnail in "trees" to view close-up

Because our three versions of the Ajax app have identical functionality at this point, we can apply the same script to all our tests. The test will be executed manually, so we kept it fairly short and made sure to flush the browser's cache in between runs. While running the test script, we recorded all the traffic, in this case using the LiveHttpHeaders plug-in for Mozilla. We then saved the HTTP session data as a text file and ran it through our script to generate a data file that could be read by a spreadsheet. We then used the spreadsheet to analyze the traffic and create a few pretty graphs. The nitty-gritty on how to achieve all these steps is given in appendix B.

Here and now, we want to see what the different flavors of Ajax have done for the performance of our application, so let's have a look at the results. Figure 4.1 com-

pares the total HTTP traffic generated by three of the QuickGallery applications, namely the non-Ajax version from chapter 2, the XML version from chapter 3, and the content-centric version from this chapter.

The first thing to notice is that viewing the close-up images takes by far the biggest toll on the network. The images weigh roughly 700 KB each. They dominate the traffic so much, in fact, that we've modified the scale of the vertical axis, in order to be able to see what's going on in the other steps of the script.

One reason often cited for not adopting Ajax—and Ajax frameworks in particular—is the weight of the additional code. As we can see in figure 4.1, the initial loading of the home page is far greater for the two Ajax apps, largely because we're loading roughly 40 KB of Prototype.js. (This data was recorded against Prototype version 1.4. Version 1.5 has grown to a little over 50 KB.) However, viewed against the traffic as a whole, it isn't making much of a difference.

In subsequent steps of the test script, we can see that the Ajax apps are making a small positive difference, with the data-centric approach typically consuming a little less bandwidth than the content-centric version. The most notable difference is in step 7, in which the non-Ajax app consumes over 100 KB, whereas the Ajax apps

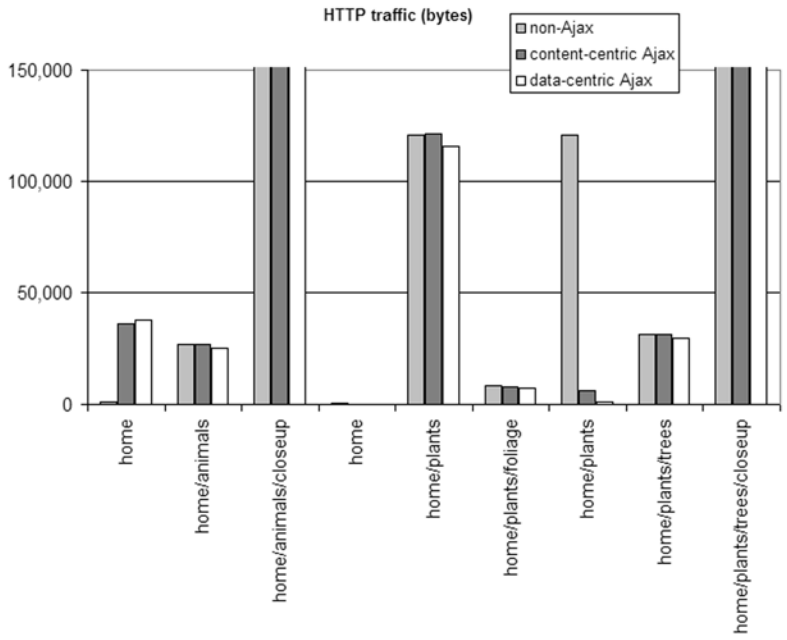

Figure 4.1 HTTP traffic generated at each step of the test script, for non-Ajax, content-centric, and data-centric Ajax versions of the QuickGallery app

consume practically nothing. Looking at the details of the logs, we can see that this is due to not having to refetch the thumbnail images for the "plants" folder.

In order to get a clearer picture of the contributions to overall traffic levels from the various types of data being sent, we've also plotted the traffic breakdown by MIME type in figure 4.2.

The pie charts in the lower half of the figure show the contribution from all media types, excepting the two large close-up images, which we've omitted again in order to make the other details show up. Even so, 90 percent or more of the total traffic comes from images. It's instructive to note how small the program-

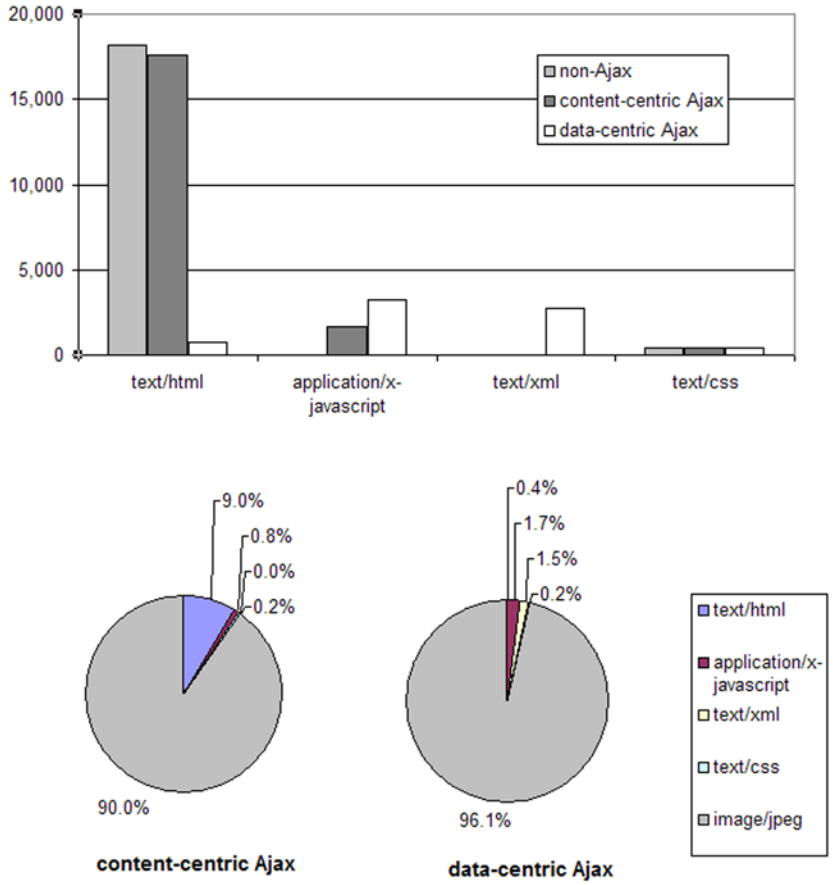

Figure 4.2 Breakdown of HTTP traffic by MIME type

matic elements of the Ajax app are in the face of broadband-sized media such as high-resolution images, videos, and audio.

The bar chart in the top half of figure 4.2 shows the relative makeup of the three apps' traffic, once we've taken the images out of the equation. It's interesting to note that our content-centric Ajax application generates almost as much HTML as the non-Ajax app, although we might expect a larger difference if the design of the application weren't so spartan. Certainly, comparing the HTML generated by the content-centric app against the XML generated by the data-centric app, we can see that the data-centric app is making more efficient use of the network when transmitting the navigational information. In order to do so, it requires roughly twice as much JavaScript code as the content-centric app.

So, in terms of overall impact on the network, how do the three solutions stack up? Figure 4.3 plots the cumulative HTTP traffic for each application.

The clear take-home message from this picture is that Ajax is good for the network. Our simple nine-step test script equates to only a minute or so of use of the

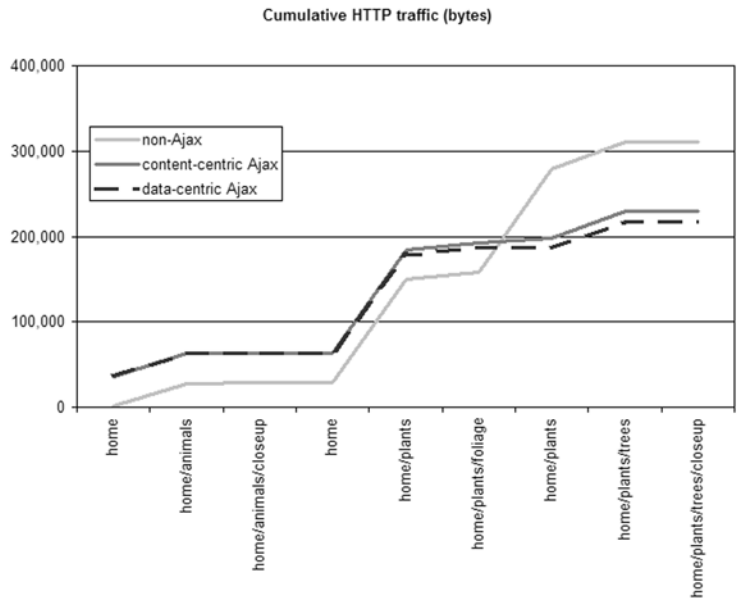

Figure 4.3 Cumulative use of bandwidth by the Ajax and non-Ajax QuickGallery applications

QuickGallery application, and we've already made a significant net saving in bandwidth by step 7. Remember, at that point, the non-Ajax application reloaded all the thumbnail images, whereas the Ajax apps didn't. If we've been looking for a clear signal to adopt Ajax as we take our development forward, this is it.

As we said earlier, there is plenty of folklore about Ajax network performance floating around on the Internet. Looking at the numbers for ourselves, we've established that Ajax is good for bandwidth in our particular application. We've also confirmed the story that data-centric solutions make more efficient use of the network than content-centric solutions (see figure 4.2). However, we can also see from figure 4.3 that in the overall picture, these savings aren't worth a great deal. This also sheds some light on the often-quoted adage that XML is a nasty, bloated data format. We aren't saying that it isn't bloated, but the bloat doesn't figure much in the overall scheme of things.

So, do we have a clear winner in terms of bandwidth performance? It seems not—we have to call this round a tie, or a very narrow victory for the data-centric version of QuickGallery at the best.

Before we move on, we ought to stress that this is a verdict about the QuickGallery application, not about data- and content-centric Ajax in general. We don't wish to add to the folklore that's out there, and we urge you to measure your own application's performance using the tools that we describe in appendix B and take things from there. Now let's move on to the final criterion for comparing the different styles of Ajax, so that we can arrive at a decision as to which one we'll use when we develop extra functionality into the QuickGallery app.

4.3.4 Looking for future-proof solutions

We've seen how our current content-centric and data-centric Ajax apps stack up against the non-Ajax app, and against each other, but we have to bear in mind that these are little more than prototypes of the all-singing, all-dancing QuickGallery that we want to go on to create. We have big plans for our app, and a to-do list as long as your arm, so we need to consider whether the two types of Ajax will be able to grow with us.

Our requirements are somewhat vague at the moment, but we do know that we want to be able to attach metadata to our images and sort the folder contents using this metadata. We also want to loosen the mapping between the navigation of the images and the underlying filesystem, so that we can display "virtual folder" contents based on search criteria. We also might want to be able to show more than one folder's contents side by side. Storing the metadata and running the searches are mostly server-side issues, but we want to be able to edit the metadata

and rearrange folder contents in the browser. To satisfy these requirements, we can see that it would be useful to have some sort of model of the folder tree held in the JavaScript layer. The data-centric approach lends itself more readily to maintaining such a model, so we find ourselves leaning in that direction when we consider this issue.

This sort of discussion can be very open-ended, and it can be hard to determine how easy or difficult the unimplemented features will be, based on choices that we might make. We can, however, look at our experience in getting this far. The content-centric application was certainly easier to write, but we already ran into issues with wanting to update multiple DOM elements from a single Ajax request. While we found a workaround by updating the sidebar and breadcrumb trail using scripts, this was something of a kludge, and it only worked because the content of the secondary DOM elements was so simple. If we implement the ability to view multiple folders at once, we might want to update several thumbnail windows at once. Prototype's Ajax helpers solve the multiple update problem in simple cases, but only by falling back on script- or data-centric Ajax for the secondary updates.

All other considerations—ease of use, fit to the Prototype.js library, and performance of the network traffic—have either pointed us toward the content-centric model or come out neutral. This is the only major obstacle to adopting a content-centric approach as we go forward, so can we see a way of getting around these problems? One possibility is to extend Ajax.Updater to support refreshing multiple DOM elements from a single response. While this will entail some extra development work, Prototype.js provides a very good mechanism for extending existing objects, so the overhead shouldn't be too large.

So we have a decision. For this project, we're adopting the content-centric approach. The decision was fairly close, and our aim here is not to promote content-centric Ajax as the only solution for all problems. Rather, we hope we've shown the process by which we've made the decision, and the range of factors that we've taken into account.

4.4 Summary

In this chapter, we looked at Prototype's advanced Ajax classes and their support for the content-centric style of Ajax. We also explored some of the features that provide secondary script- and data-centric support. We applied these classes to our QuickGallery application and noticed a significant improvement in developer productivity over the data-centric approach that we used in chapter 3.

In deciding which approach to use as our application development goes forward, we looked at a range of criteria. By analyzing the HTTP traffic, we were able to see significant improvements over the non-Ajax application. The data-centric approach came first in only one category: performance of HTTP traffic. However, although the data-centric application made better use of the network than the content-centric one, the overall impact for our application was insignificantly small, leaving the content-centric approach as the clear way forward.

It's important to stress again that we reached this decision for this specific application. Rather than remembering the conclusion that we came to, we urge you to remember our decision-making process, and follow it in order to reach your own conclusions.

This concludes our review of Prototype.js's Ajax helper classes. In the next section of the book, we're going to look at the Scriptaculous library, and the ways in which it can enhance the usability of our application.

WEB PROGRAMMING

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 54499

MANNING

Ebook at: manning.com/crane3
$44.99 US/$58.99 Canada