

Practical Prototype and script.aculo.us

Copyright © 2008 by Andrew Dupont

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-919-8

ISBN-10 (pbk): 1-59059-919-5

ISBN-13 (electronic): 978-1-4302-0502-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editors: Clay Andres, Tony Campbell, Jason Gilmore, Chris Mills

Technical Reviewer: Aaron Gustafson

Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell,

Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann,

Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh

Project Manager: Beth Christmas

Copy Editor: Damon Larson

Associate Production Director: Kari Brooks-Copony

Production Editor: Laura Esterman

Compositor: Linda Weidemann, Wolf Creek Press

Proofreader: Erin Poe

Indexer: Broccoli Information Management

Artist: April Milne

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at <http://www.apress.com/info/bulksales>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com>.



Ajax: Advanced Client/Server Communication

By now, you're almost certainly familiar with *Ajax* as a buzzword. Technically, it's an acronym—Asynchronous JavaScript and XML—and refers specifically to JavaScript's `XmlHttpRequest` object, which lets a browser initiate an HTTP request outside the confines of the traditional page request.

Yawn. The technology isn't the exciting part. Ajax is huge because it pushes the boundaries of what you can do with a web UI: *it lets you reload part of a page without reloading the entire page*. For a page-based medium like the Web, this is a seismic leap forward.

Ajax Rocks

`XmlHttpRequest` (XHR for short) is a JavaScript interface for making arbitrary HTTP requests. It lets a developer ask the browser to fetch a URL in the background, but without any of the typical baggage of a page request—the hourglass, the new page, and the re-rendering.

Think of it as an HTTP library for JavaScript, not unlike Ruby's `Net::HTTP` class or PHP's `libcurl` bindings. But because it lives on the client side, it can act as a scout, marshaling requests between client and server in a much less disruptive way than the typical page request.

The difference is crucial—the user has to wait around for a page request, but XHR doesn't. Like the acronym says, Ajax allows for *asynchronous* communication—the JavaScript engine can create a request, send it off, and then do other things until the response comes back. It's far better than making your *users* do other things until the response comes back.

Ajax Sucks

It's not all sunshine and rainbows, though. Ajax is much easier to talk about than it is to *do*.

The problem that afflicts JavaScript in general applies to Ajax in particular: the XMLHttpRequest object has its own set of bugs, inconsistencies, and other pitfalls from browser to browser. Created by Microsoft and first released as part of Internet Explorer 5, the XHR object gained popularity once it was implemented by the other major browser vendors—even though there was no formal specification to describe how it ought to work. (The W3C has since started an XHR specification, currently in “Working Draft” status.)

For this reason, it's painful and frustrating to work with XHR without some sort of wrapper library to smooth out the rough edges. Prototype takes the awkward, unintuitive API of XMLHttpRequest and builds an easy-to-use API around it.

Prototype's Ajax Object

Let's set up an environment to play around with Ajax. In a text editor, create a file named `ajax.js` and place some JavaScript content inside. This will be the file we load with Ajax (see Listing 4-1).

Listing 4-1. *The ajax.js File*

```
alert('pancakes!');
```

Create a directory for this file and save it.

Caution Since Ajax is an HTTP request interface, these examples require a web server to communicate with. Opening these examples straight from the local disk (using the `file:` protocol) will yield mixed results. Try running them on a local installation of Apache—or on space you control on a remote web server.

Now we need a page to make the Ajax request *from*. Create an empty HTML page, call it `index.html`, and save it in the same directory as `ajax.js`. Listing 4-2 shows the `index.html` file.

Listing 4-2. *The index.html File*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Blank Page</title>

    <script src="prototype.js" type="text/javascript"></script>
  </head>

  <body>
    <h1>Blank Page</h1>
  </body>
</html>
```

Notice how we load Prototype by including it in the head of our document via a `script` tag. You'll need to place a copy of `prototype.js` in the same directory as `index.html`.

Now open `index.html` in Firefox. We'll use Firefox for these examples so that we can execute commands on the fly in the Firebug interactive shell. Make sure the Console tab is focused, as shown in Figure 4-1.

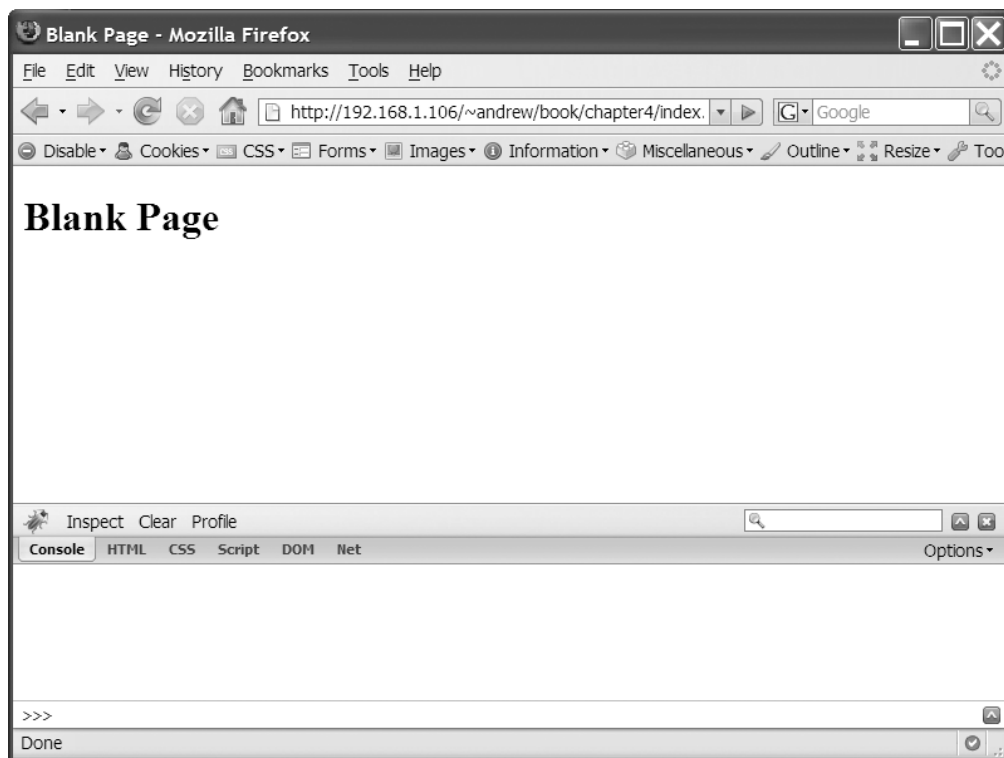


Figure 4-1. Firebug is open to the console tab at the bottom of the screen.

Ajax.Request

Now type the following into the shell:

```
new Ajax.Request('ajax.js', { method: 'get' });
```

You should see the dialog shown in Figure 4-2.

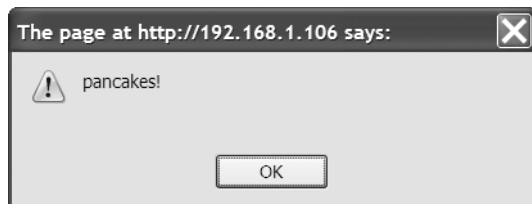


Figure 4-2. This dialog came from our external JavaScript file.

The `Ajax.Request` call fetched our external file and evaluated the JavaScript we placed inside. This simple example teaches you several things about `Ajax.Request`:

- It's called as a constructor (using the `new` keyword).
- Its first argument is the name of the URL you want to load. Here it's a relative URL because the file we want to load is in the same directory; but you can also use an absolute URL (one that begins with a forward slash).

Caution Keep in mind that this URL can't begin with `http` because of the same-domain policy of Ajax—even if the URL points internally.

- Its second argument is an object that can contain any number of property/value pairs. (We'll call this the options argument.) Prototype uses this convention in a number of places as a way of approximating named arguments. In this example, we're specifying that the browser should make an HTTP GET request for this file. We only need to specify this because it's overriding a default—if you omit the `method` option, `Ajax.Request` defaults to a POST.
- The JavaScript we placed in `ajax.js` was evaluated *automatically*, so we know that `Ajax.Request` will evaluate the response if it's served up as JavaScript. Web servers typically give JS files a MIME type of `text/javascript` or `application/x-javascript`; Prototype knows to treat those types (and a handful of others) as JavaScript.

Now let's add to this example. Type the same line as before, but with an extra property in the options argument:

```
new Ajax.Request('ajax.js', { method: 'get',  
  onComplete: function() { alert('complete'); }  
});
```

Tip You can switch the Firebug console to multiline input by clicking the button at the far right of the command line.

Figure 4-3 shows the results. This time, you'll see two dialogs: the original, “pancakes!” and the one inside the line highlighted in the previous code block, “complete.”

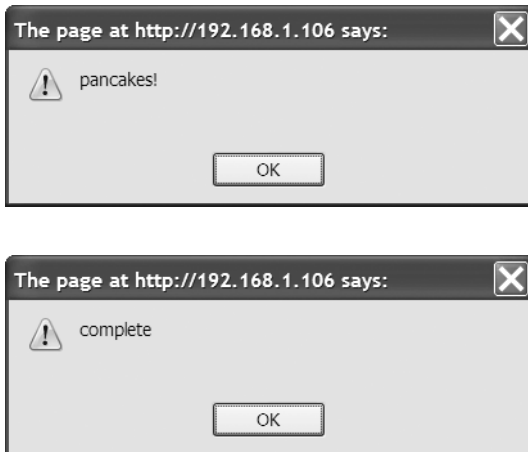


Figure 4-3. *These two dialogs appear in sequence.*

So, by adding just a little code to this example, you’ve learned two more things:

- The `onComplete` option is a new property in our options object. It sets up a *callback*—a function that will run at a certain point in the future. An Ajax request keeps the browser updated on its progress, triggering several different “ready states” along the way. In this case, our `onComplete` function will be called when the request is complete.
- The “pancakes!” dialog appears before the “complete” dialog, so you can deduce that the `onComplete` function is called *after* the response is evaluated.

Let’s use another callback. Replace `onComplete` with `onSuccess` (see Figure 4-4):

```
new Ajax.Request('ajax.js', { method: 'get',  
    onSuccess: function() { alert('success'); }  
});
```

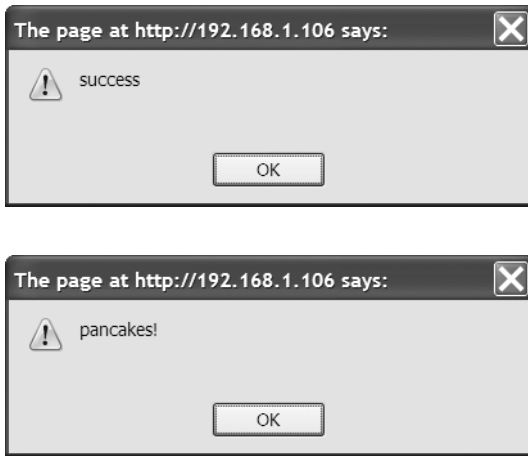


Figure 4-4. *These two dialogs appear in sequence.*

Figure 4-4 is subtly different than Figure 4-3. Like before, you'll see two dialog boxes—but this time the “pancakes!” dialog comes last. So, you can assume the following:

- The `onSuccess` option is a callback that will be run if the request is a success. If there's a failure of some kind (a 404 error, a communication error, an internal server error, etc.), its companion, `onFailure`, will get called instead.
- Since we saw the callback's alert dialog first, we know that `onSuccess` and `onFailure` are called before the remote JavaScript file is evaluated, and also before `onComplete`. True to its name, `onComplete` is called as the *very last thing* Ajax.Request does before it punches its timecard. But it decides between calling `onSuccess` or `onFailure` as soon as it knows the outcome of the request.

We can request any kind of file with Ajax—not just JavaScript files. To prove it, rename `ajax.js` to `ajax.txt` and try this (see Figure 4-5):

```
new Ajax.Request('ajax.txt', { method: 'get',  
  onSuccess: function(request) { alert(request.responseText); }  
});
```

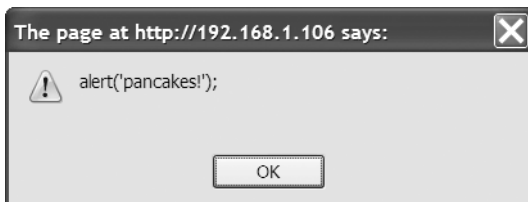


Figure 4-5. *Our new dialog contains some familiar text.*

You just learned two more things from Figure 4-5:

- Because our “pancakes!” dialog didn’t appear, we know that the response was not evaluated as JavaScript—because it was served as an ordinary text file.
- Callbacks like `onSuccess` are passed the browser’s native `XmlHttpRequest` object as the first argument. This object contains several things of interest: the `readyState` of the request (represented as an integer between 0 and 4), the `responseText` (plain-text contents of the requested URL), and perhaps the `responseXML` as well (a DOM representation of the content, if it’s served as HTML or XML). That’s how we were able to display the contents of `ajax.txt` in our dialog.

Here’s where it all comes together—since we can fetch a fragment of HTML from a remote file, we can update the main page incrementally by dropping that fragment into a specific portion of the page. This is such a common task that Prototype has a subclass for it.

Ajax.Updater

Prototype’s `Ajax.Updater` does exactly what you think it does: it “updates” a portion of your page with external content from an Ajax request.

To demonstrate this, let’s add an empty container to our `index.html` file.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Blank Page</title>

    <script src="prototype.js" type="text/javascript"></script>
  </head>

  <body>
    <h1>Blank Page</h1>
    <div id="bucket"></div>
  </body>
</html>
```

Now we can request an external HTML file and direct the browser to place its contents into the div we just created. So let's create a file called `ajax.html`, as shown in Listing 4-3.

Listing 4-3. *The `ajax.html` File*

```
<h2>(actually, it's not blank anymore)</h2>
```

This isn't a full HTML file, you'll notice—since we'll be inserting this content into a fully formed page, it should just be an HTML fragment.

Now reload `index.html` in Firefox. You won't see the div we created, of course, because there's nothing in it yet. Type this into the Firebug console:

```
new Ajax.Updater('bucket', 'ajax.html', { method: 'get' });
```

In Figure 4-6, you'll see our once-empty div chock-full of content!

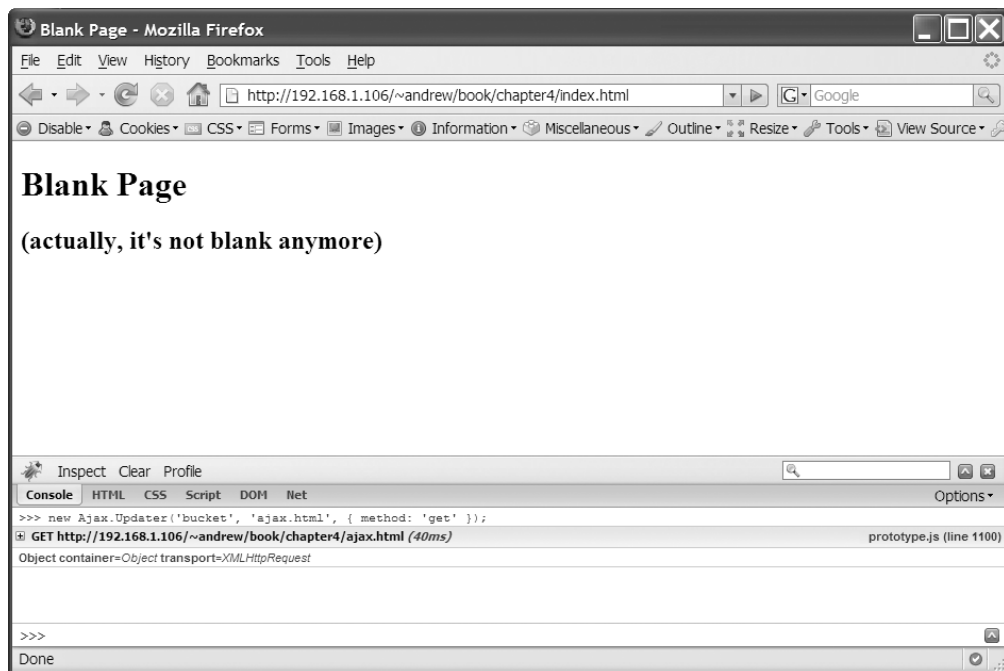


Figure 4-6. *Our `h1` is no longer alone on the page.*

This line of code reads almost like a sentence: *Using Ajax, update the bucket element with the contents of ajax.html*. It introduces you to more new things:

- `Ajax.Updater` works a lot like `Ajax.Request`. But it's got an extra argument at the beginning: the element to be updated. Remember what you learned in Chapter 2: any function that takes a DOM node can also take a string reference to that node's ID. We could just as easily have used `$('bucket')` (or a native DOM call like `document.getElementsByTagName('div')[0]`) as the argument instead of `'bucket'`.
- Just like `Ajax.Request`, `Ajax.Updater` takes an options hash as its final argument. It supports all the options we've covered already, plus a few new ones, which we're about to look at.

Now press the up arrow key at the Firebug command line to bring up the statement you just typed. Run it again (see Figure 4-7).

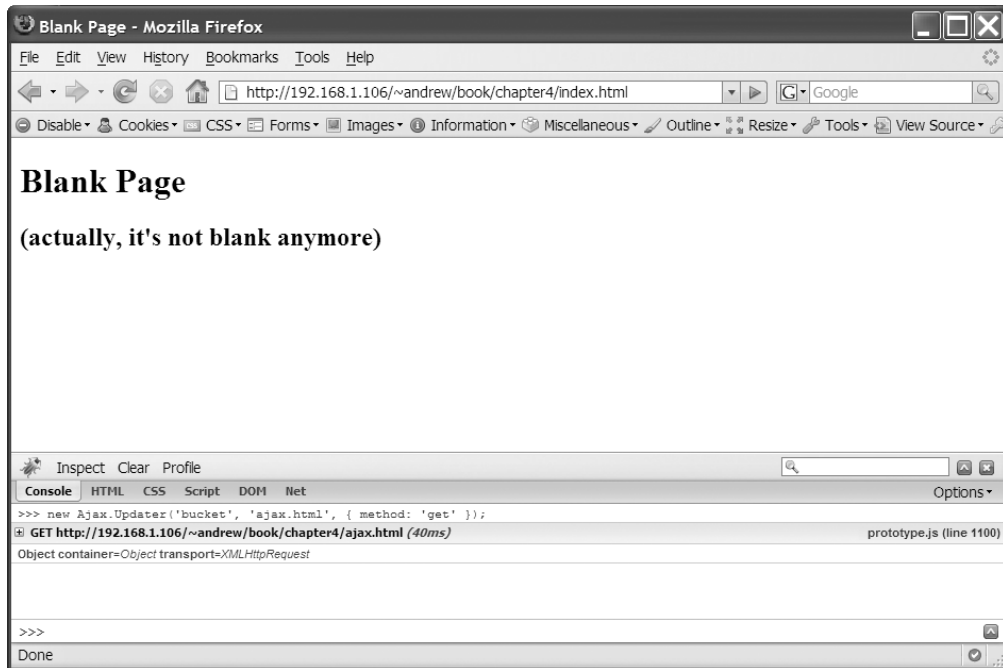


Figure 4-7. *Isn't this the same as the last?*

Nothing changed between Figures 4-6 and 4-7. Well, that's not true—something changed, but you didn't notice because the old content was identical to the new content. Every time you call `Ajax.Updater` on an element, it will *replace* the contents of that element.

You can change this behavior with one of `Ajax.Updater`'s options: `insertion`. If present, the updater object will add the response to the page without overwriting any existing content in the container.

The `insertion` property takes one of four possible values: `top`, `bottom`, `before`, or `after`. Each one inserts the content in the described location, relative to the container element: `top` and `bottom` will insert *inside* the element, but `before` and `after` will insert *outside* the element.

So let's try appending the response instead. Type this into your console:

```
new Ajax.Updater('bucket', 'ajax.html', { method: 'get', insertion: 'bottom' });
```

Although it's a bit longer, this line of code also reads like a sentence: *Using Ajax, get the contents of ajax.html and insert them at the bottom of the bucket element.*

Run this code. Then run it again, and again, and again. Each time you'll see an extra `h2` tag on the page, as shown in Figure 4-8.

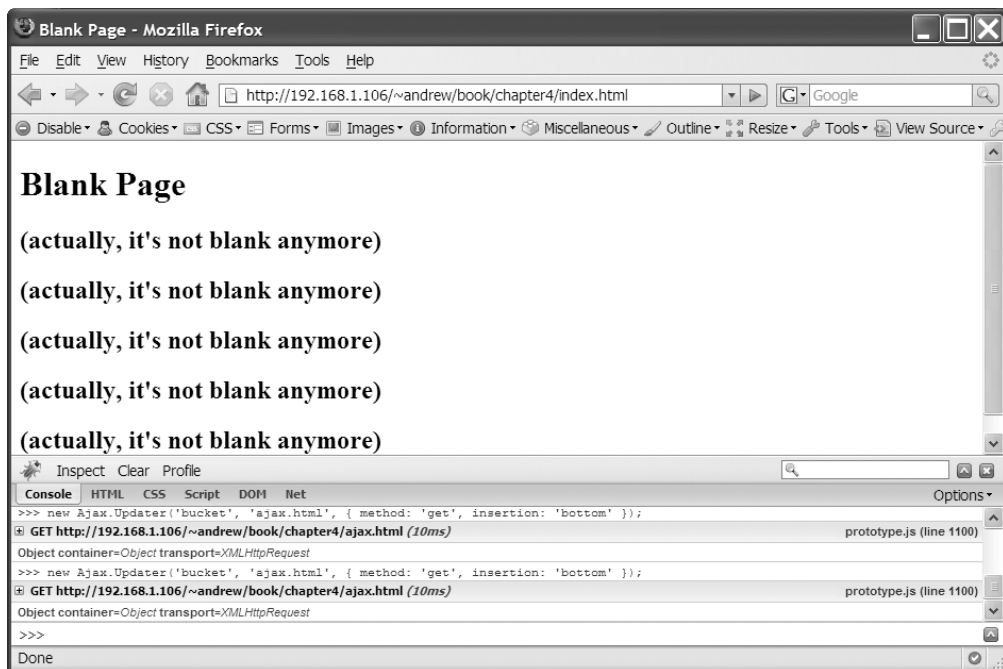


Figure 4-8. The `h2`s are starting to reproduce like mad.

This is pretty cool stuff. It's a shame you have to run this code every single time, though. You could pay someone to sit at your desk and enter this line into the Firebug console over and over—but it's probably easier to use `Ajax.PeriodicalUpdater`.

Ajax.PeriodicalUpdater

Just as Updater builds on Request, PeriodicalUpdater builds on Updater. It works like it sounds: give it a URL, an element to update, and a time interval, and it will run an Ajax.Updater at that interval for the life of the page—or until you tell it to stop.

There are tons of applications for a repeating Ajax request: imagine the client side of a chat application asking the server if anyone's spoken in the last 10 seconds. Imagine a feed reader that checks every 20 minutes for new content. Imagine a server doing a resource-intensive task, and a client polling every 15 seconds to ask how close the task is to completion.

Rather than dispatch an Ajax request as the result of a user action—a click of a button or a drag-and-drop—these examples set up a request to run automatically, thereby saving the user the tedious task of manually reloading the page every time.

Let's try it. Reload `index.html` and run this command in the console:

```
new Ajax.PeriodicalUpdater('bucket', 'ajax.html', {  
  method: 'get', insertion: 'bottom', frequency: 5  
});
```

Right away we see our first `h2` element. Then, 5 seconds later, we see another. Then another. Now they're reproducing with no help from us.

So here's what you've probably figured out about PeriodicalUpdater:

- It takes the same basic arguments as `Ajax.Updater`, but it also accepts a `frequency` parameter in the options object, allowing you to set the number of seconds between requests. We could have omitted this parameter and left the frequency at the default 2 seconds—but then we wouldn't have had the occasion to talk about it.
- At the specified interval, `Ajax.PeriodicalUpdater` will create its own instance of `Ajax.Updater`—passing it the element to update, the URL to request, and any relevant options. For example, the `insertion` parameter is being passed to `Ajax.Updater`, which is why new content is being added to the bottom of `div#bucket` instead of replacing the old stuff.

OK, these `h2`s are getting on my nerves. Reload the page.

Controlling the Polling

When you set up a `PeriodicalUpdater`, you don't necessarily want it to keep running until the end of time. Prototype gives us a couple of tactics to regulate this constant flow of requests.

To demonstrate the first, we'll have to interact with the `PeriodicalUpdater` instance. So let's run our most recent line of code again, making sure to assign it to a variable:

```
var poller = new Ajax.PeriodicalUpdater('bucket', 'ajax.html', {  
  method: 'get', insertion: 'bottom', frequency: 5  
});
```

We could have been doing this all along, but only now do we need to refer to the `Ajax` object in subsequent lines of code.

The familiar steady stream of `h2s` is back, inexorably marching down the page like textual lemmings. But this time we can make them stop:

```
poller.stop();
```

Spend a few seconds staring at the screen, nervously wondering if you truly shut it all down. You'll eventually realize that the `h2s` have stopped, and no more `Ajax` requests are being logged to the Firebug console.

It's really that simple: `PeriodicalUpdater` has an instance method named `stop` that will put all that periodical updating on hold. There's a predictable complement to `stop`, and we'll use it right now to turn the `h2s` back on:

```
poller.start();
```

Their respite was short-lived—the `h2s` are back and growing in number every 5 seconds. Calling `start` on a stopped `PeriodicalUpdater` works a lot like creating a new one: the request is run immediately, and then scheduled to run again once the specified interval of time passes.

There's one more flow control strategy we can employ. It's called *decay*, but it's nowhere near as gross as it sounds—think atoms, not carcasses.

Let's reload `index.html` one more time and add the `decay` parameter to our options object:

```
var poller = new Ajax.PeriodicalUpdater('bucket', 'ajax.html', {  
  method: 'get', insertion: 'bottom', frequency: 5, decay: 2  
});
```

It will take a little longer to realize what's going on this time. Just like before, the first `Ajax` request is dispatched immediately. Then there's another request 5 seconds later. Then . . . wait. That time it felt more like 10 seconds. And now it's even longer. Is this thing *slowing down*?

After a few more cycles, you'll be able to figure out what's going on.

It is, in fact, slowing down. Our decay parameter is causing the interval between requests to double each time. (5 seconds, then 10, 20, 40, 80, etc.) If we changed decay to 3, the interval would be tripled each time. In mathematics, this is called *exponential decay*. It has many applications across all fields of science, but here we’re using it to make web pages *awesome*. The default value for decay is 1—that is, by default there *is no* decay.

But *why* is it slowing down? Because it’s getting the same response every time. `PeriodicalUpdater` keeps track of this, comparing the latest response to the previous response each time the updater runs. If the contents are different, all proceeds as normal; if the contents are identical, the interval gets multiplied by the decay parameter and the result is used to schedule the next updater. (In this example, of course, we’re requesting a static HTML file, so each request is identical to the previous.) If, after the interval is lengthened, a fresh response comes back, it snaps back to the frequency that was originally set.

So `PeriodicalUpdaters` can be started, stopped, and decayed, abiding by your exacting rules of flow control. You’ll need these rules someday. You probably didn’t feel any dread at the prospect of HTML elements that reproduce infinitely, but you will feel dread when the server hosting your web app starts getting hit every 15 seconds by every single client using it. Responsiveness is good, and periodic client-server communication is good—but these benefits will eventually clash with the practical constraints of bandwidth and processing power. Knowing when to poll, when not to poll, and when to poll *less often* can be the antidote to the typical “chattiness” of Ajax-driven applications.

Advanced Examples: Working with Dynamic Content

We’ve already looked at a handful of simple examples of what Prototype’s Ajax objects can do. But simple examples are boring. Let’s get our feet wet.

Increasing the complexity means we’ll have to introduce server-side scripting to the mix. These examples will use PHP, but the concepts are applicable no matter what your architecture.

Example 1: The Breakfast Log

Most of your Ajax calls will involve dynamic content, rather than the HTML and text files we’ve been using—the response will vary based on the data you send. You’re probably already familiar with GET and POST—the two HTTP methods for sending data to the server—from working with HTML forms. Ajax can use either method to submit data.

For this set of examples, we'll be creating a "blog." If you don't know what a blog is, you're behind the times, my friend; it's short for "breakfast log," and it's a minute-by-minute account of which breakfast foods you've consumed on which dates and times. The trend is spreading like wildfire: at least half a dozen people on earth have breakfast logs.

The Server Side

We'll start with the server side, so that our page will have something to talk to. Create a file called `breakfast.php` (as shown in Listing 4-4) and put it in the same directory as `index.html`.

Listing 4-4. *The breakfast.php File*

```
<?php
// make a human-readable date for the response
$time      = date("g:i a \o\\n F j, Y", time());

$food_type = strip_tags($_REQUEST['food_type']);
$taste     = strip_tags($_REQUEST['taste']);
?>

<li>At <strong><?= $time ?></strong>, I ate <strong><?= $taste ?>
<?= $food_type ?></strong>.</li>
```

I've highlighted the lines that reference `$_REQUEST`, the global variable for looking up any query parameters given to a script. This script expects to receive two crucial pieces of data: the kind of food I ate and its taste quality. For now, we won't send the time of the meal—we'll simply assume that the breakfast logger ("blogger" for short) is posting from his or her kitchen table, mouth full of French toast. This lets us take the shortcut of using the server's time rather than relying on the client to provide it.

Our script takes the provided values for food and taste and strips them of HTML using PHP's `strip_tags` function. (Ordinarily, it would also save the values to a database, but that's not important for what we're doing here.) Then it prints a small fragment of HTML to describe to the browser what has just happened.

Let's make sure our script works. Open a browser and navigate to wherever you put `breakfast.php` on your server. But remember, we've got to tell it what we ate and how delicious it was. So we need to add a query string to the end of the URL. Yours should look something like this:

`http://your-server.dev/breakfast.php?food_type=waffles&taste=delicious`

Press return, and you ought to see your HTML fragment in the browser window, as shown in Figure 4-9.

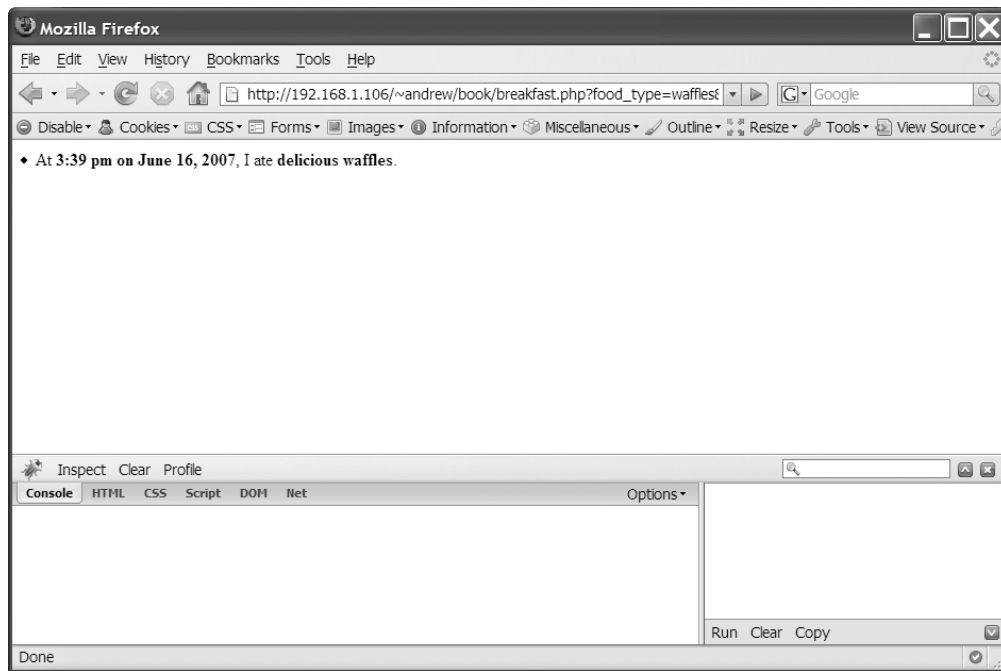


Figure 4-9. *The HTML fragment that represents our delicious meal*

No errors! Emboldened by this programming victory, let's go back to `index.html` to make it look more like a breakfast log.

The Client Side

Our HTML page is no longer generic—it's got a purpose! Let's make it friendlier. Make these changes to `index.html`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Andrew's Breakfast Log</title>

    <script src="prototype.js" type="text/javascript"></script>
  </head>
```

```

<body>
  <h1>Andrew's Breakfast Log</h1>
  <ul id="breakfast_history"></div>
</body>
</html>

```

It's still ugly and sparse, but at least it's got a human touch now.

We're going to keep a list on this page, so let's treat it as such. We've changed our container div to a ul, a proper container for lis, and given it a more descriptive ID.

Now we're ready to record our meals for posterity! Reload index.html in Firefox, and then type this into the Firebug console:

```

new Ajax.Updater('breakfast_history', 'breakfast.php', { method:'get',
  parameters: { food_type: 'waffles', taste: 'delicious' }
});

```

You should recognize the highlighted line—we're sending these name/value pairs along with our request. Our script gets the message, saves it to a database (presumably), and then gives us some HTML to put on the page.

Also, notice how we've removed the method parameter from the options. We could explicitly set it to "post", but since that's the default we're better off omitting it altogether.

Run this code. The result should look like Figure 4-10.

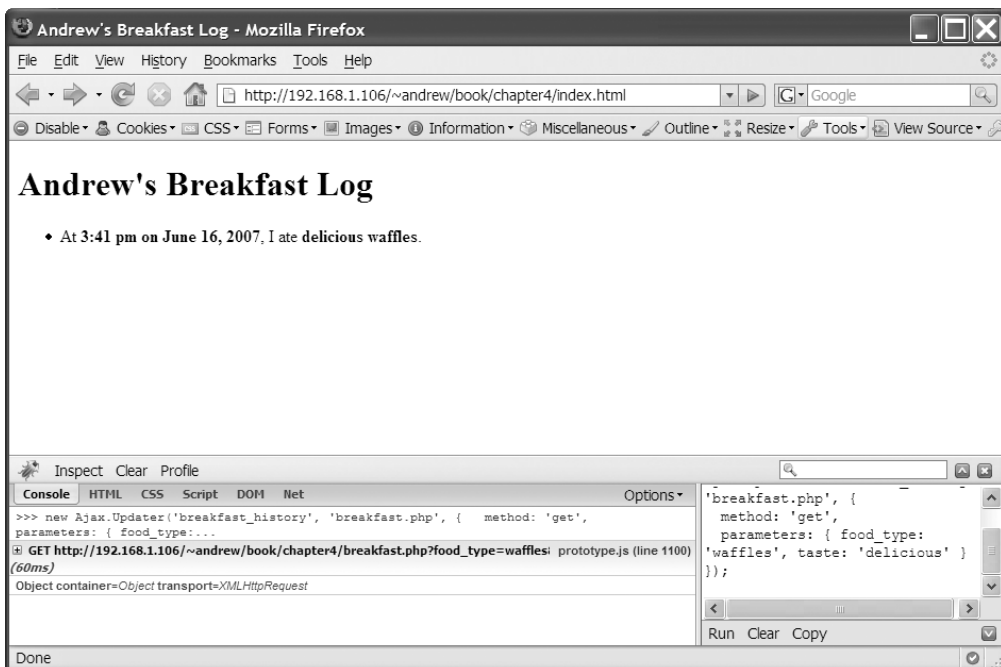


Figure 4-10. The fragment from the previous figure has been placed on the page.

Since Firebug logs all Ajax requests, you can see for yourself. Near the bottom of your console should be a gray box containing the URL of the request; expand this box to view all the request's details, as depicted in Figure 4-11.

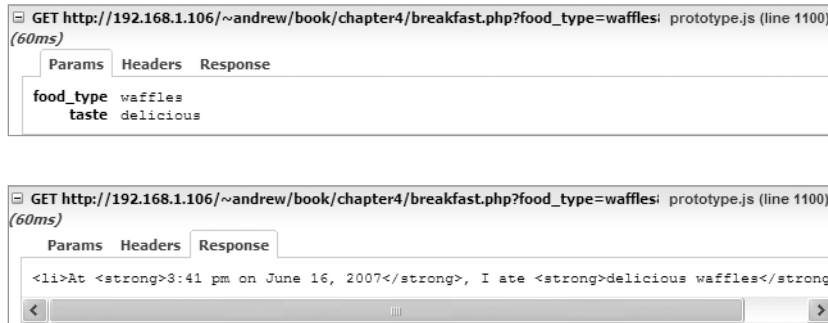


Figure 4-11. The details of our Ajax request

That was fun. Let's try it again—run the exact same command in the console (see Figure 4-12).

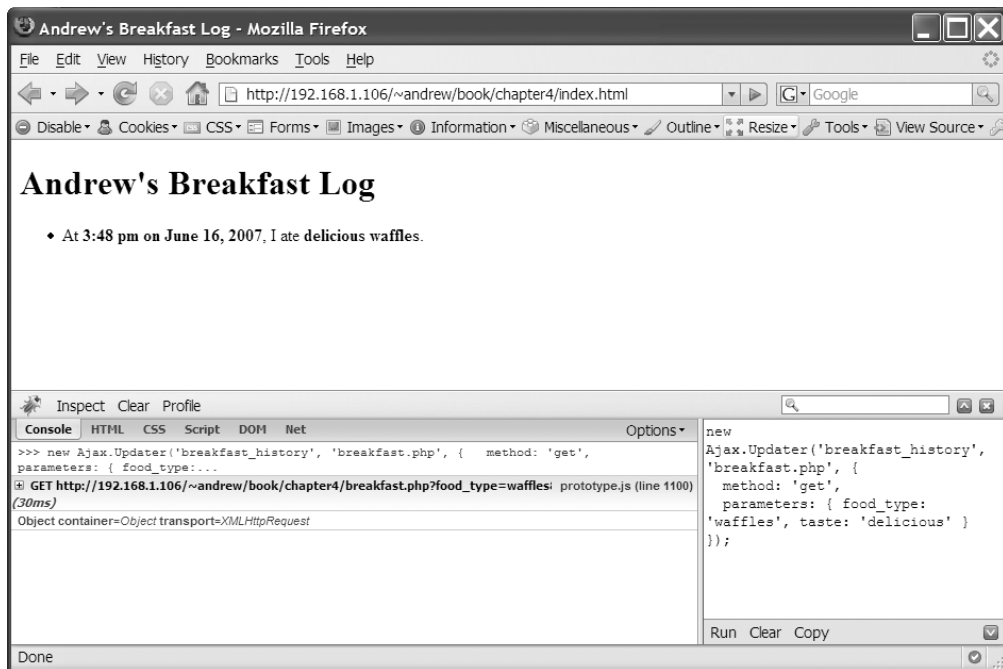


Figure 4-12. What happened to the first one?

Figure 4-12 is not quite what we expected. The time is different, so the content got replaced properly. But we don't want to *replace* the contents of `ul#breakfast_history`; we want to *add* to what's already there.

Typically, breakfast log entries are arranged so that the most recent is first. So let's change our Ajax call so that new entries are appended to the top of the container:

```
new Ajax.Updater('breakfast_history', 'breakfast.php', {
  insertion: 'top', method: 'get',
  parameters: { food_type: 'waffles', taste: 'delicious' }
});
```

Run this code and you'll see your new entry added to the top of the list, as in Figure 4-13. Each time you run this code, in fact, a new `li` will be added to the top of your `ul` container.

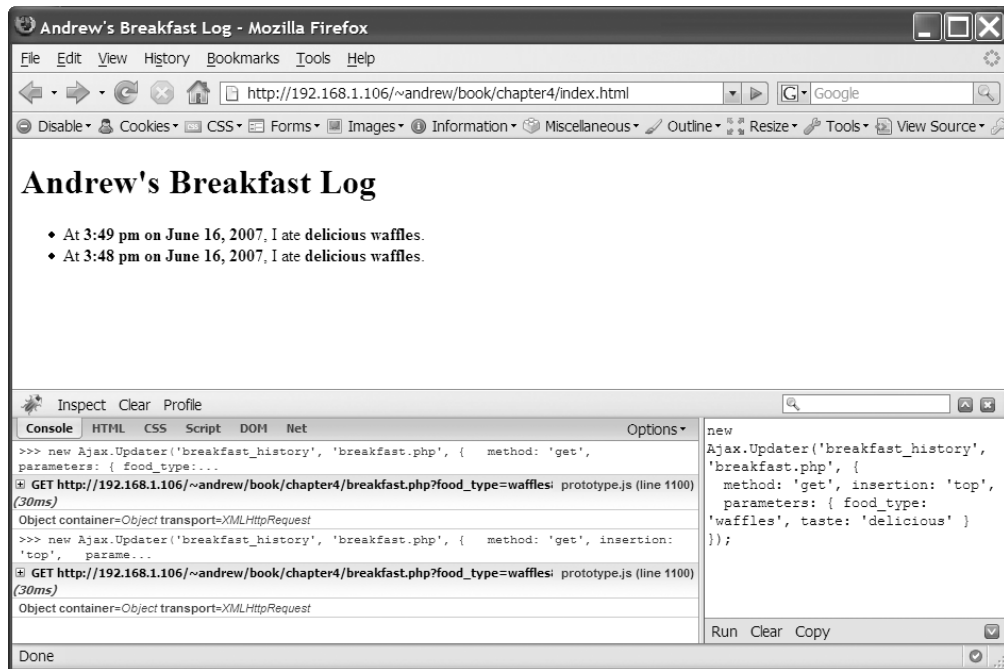


Figure 4-13. *New entries appear at the top.*

Handling Errors

Our `breakfast.php` script works, but it's not exactly battle-tested. It naively assumes that each request will have the two pieces of information it wants. What if something goes wrong? What if our scrambled eggs fail to be tabulated? We need to work out some sort of code between client and server to handle situations like these.

Actually, it's been worked out for us. Each HTTP request has a status code that indicates whether everything went well or not. The standard success response is 200, which means "OK," although most web surfers are more familiar with 404 (File Not Found), since one usually isn't shown the status code until something goes wrong.

The first digit of an HTTP status code tells you what kind of message this is going to be. Codes starting with 2 are all various forms of success codes, 3 signifies a redirect, 4 means the request was faulty somehow, and 5 means that the server encountered an error.

This is just what we need. Our script can check for the presence of `food_type` and `taste` as query parameters. If it doesn't find them, it can return an error status code instead of the typical 200. And it can use the content of the response to present a friendlier error message to the user.

PHP lets us do this rather easily.

```
<?php
// make a human-readable date for the response
$time      = date("g:i a \o\\n F j, Y", time());

if (!isset($_REQUEST['food_type']) || !isset($_REQUEST['taste'])) {
    header('HTTP/1.0 419 Invalid Submission');
    die("<li>At <strong>${time}</strong>: Whoa! Be more descriptive.</li>");
}

$food_type = strip_tags($_REQUEST['food_type']);
$taste     = strip_tags($_REQUEST['taste']);
?>
<li>At <strong><?= $time ?></strong>, I ate <strong><?= $taste ?>
<?= $food_type ?></strong>.</li>
```

The 419 error code isn't canonical—we just made it up. But Apache delivers this code just fine, and Prototype properly recognizes it as an error code.

Test this in your browser. Open up `breakfast.php` directly, just like you did before—but this time leave one of the parameters out of the URL (see Figure 4-14):

```
http://your-server.dev/breakfast.php?food_type=waffles
```

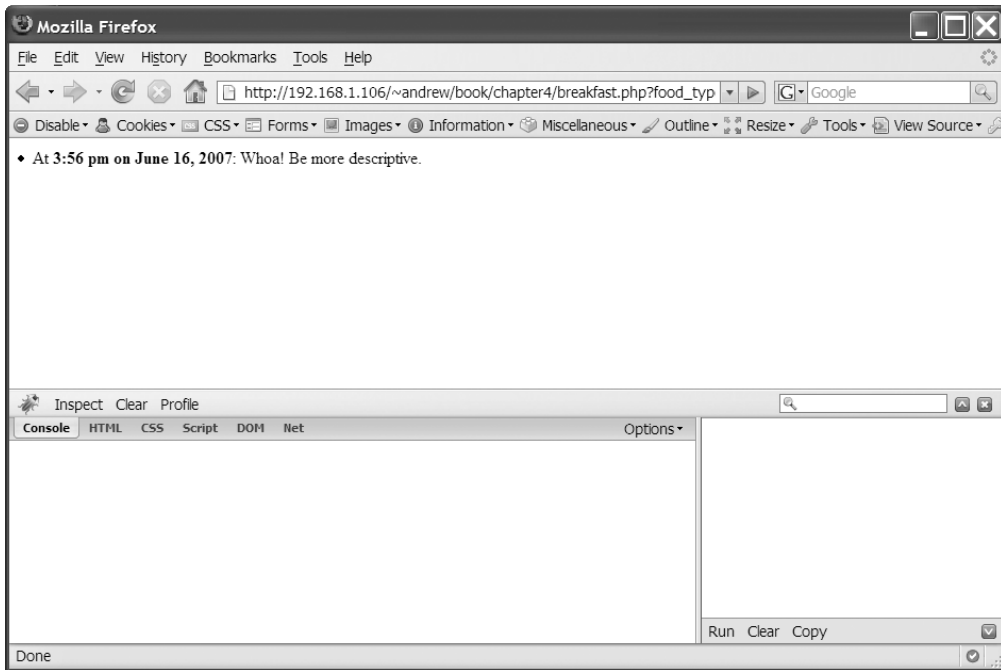


Figure 4-14. *Our error is a success!*

As expected, the response (shown in Figure 4-14) tells us that we weren't forthcoming enough about the waffles we just ate. We can't actually see the status code this way, but we can if we request the URL through Ajax instead. So go back to `index.html` and run the `Ajax.Updater` call once more—but this time remove one of the parameters from the options hash. (Figure 4-15 shows the result.)

```
new Ajax.Updater('breakfast_history', 'breakfast.php', {  
  insertion: 'top',  
  parameters: { food_type: 'waffles' }  
});
```

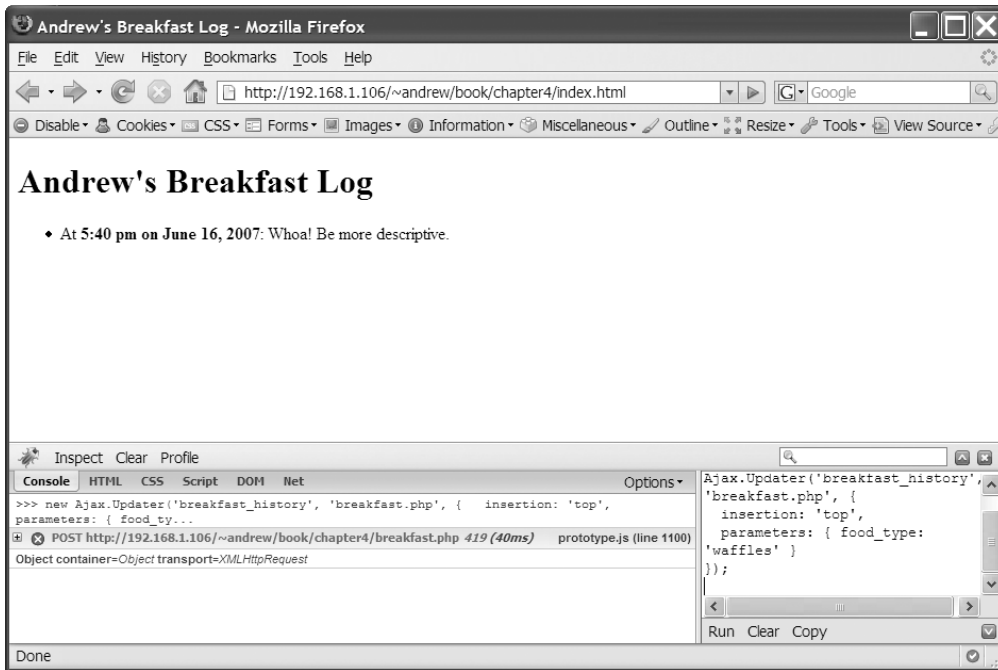


Figure 4-15. Proper error reporting

Firebug, ever helpful, shows you that something went wrong with the request—it makes the URL red and adds the status code to the end of the line, as shown in Figure 4-16.

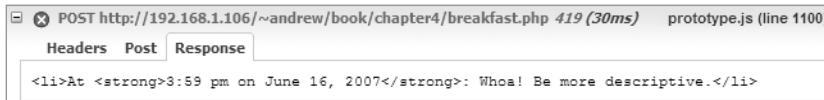


Figure 4-16. Firebug shows the status code when an error occurs.

So our omission of taste information is being reported as an error, just like we want. But our JavaScript code doesn't yet treat errors differently from successful responses. We need to separate the two if we want errors to stand out to the user.

So let's create a new `ul`, this one for errors. We can style the two containers differently and give them headings so that the user knows they're two different groups of things. Make these changes to `index.html` and view the results in Figure 4-17:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Andrew's Breakfast Log</title>
    <style type="text/css" media="screen">
      #breakfast_history {
        color: green;
        border: 1px solid #cfc;
        padding: 5px 0 5px 40px;
      }
      #error_log {
        color: red;
        border: 1px solid #edd;
        padding: 5px 0 5px 40px;
      }
    </style>

    <script src="prototype.js" type="text/javascript"></script>
  </head>

  <body>
    <h1>Andrew's Breakfast Log</h1>

    <h2>Breakfast History</h2>
    <ul id="breakfast_history"></ul>

    <h2>Errors</h2>
    <ul id="error_log"></div>
  </body>
</html>
```

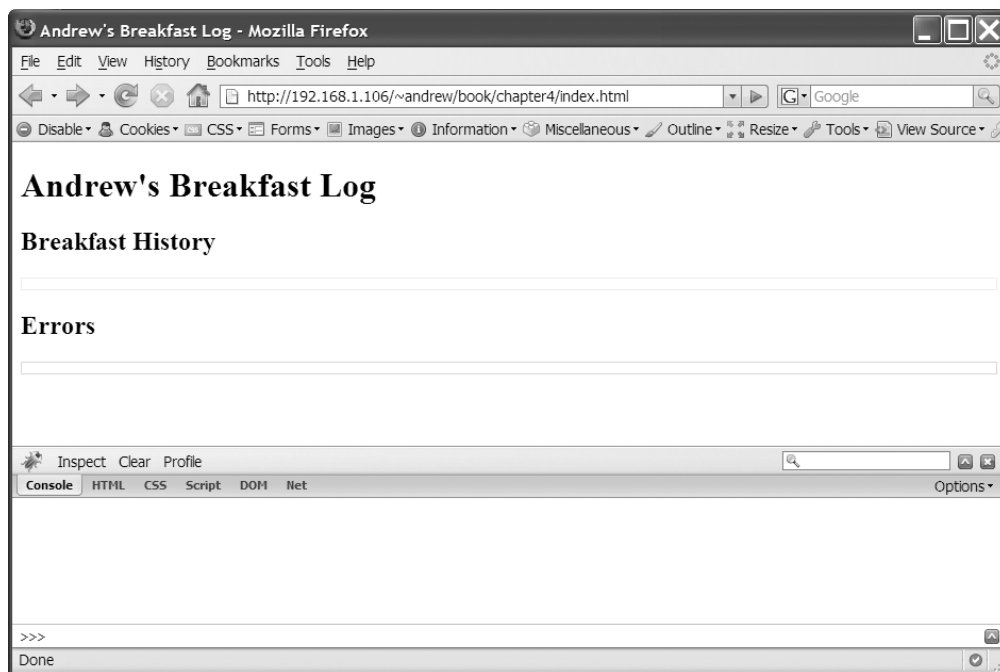



Figure 4-17. *Slightly less ugly*

We're almost there. The last part is the easiest because it's built into `Ajax.Updater`. Instead of designating one container to update, you can designate two: one for successful requests and one for unsuccessful requests. The conventions followed by HTTP status codes make it easy to figure out what's an error and what's not.

```
new Ajax.Updater({ success: 'breakfast_history', failure: 'error_log' },
  'breakfast.php', { insertion: 'top',
    parameters: { food_type: 'waffles' }
  });
```

Victory! As Figure 4-18 shows, bad requests show up bright red in the error log.

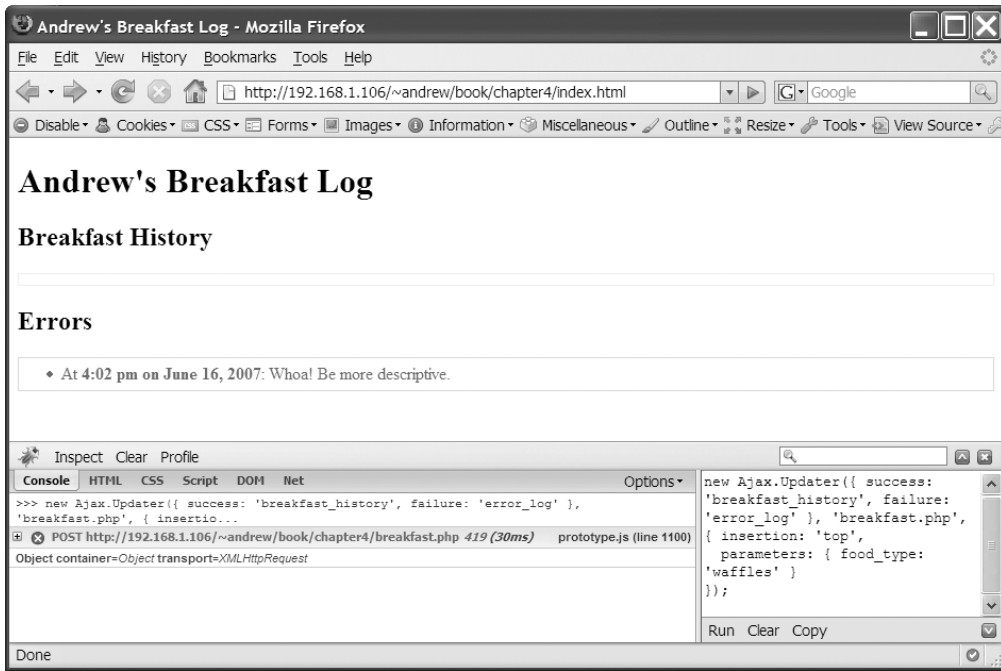


Figure 4-18. Errors now look like errors.

Now put that taste parameter back into the statement and watch your valid request appear in a pleasing green color, as shown in Figure 4-19.

```
new Ajax.Updater({ success: 'breakfast_history', failure: 'error_log' },
'breakfast.php', { insertion: 'top',
  parameters: { food_type: 'waffles', taste: 'delicious' }
});
```

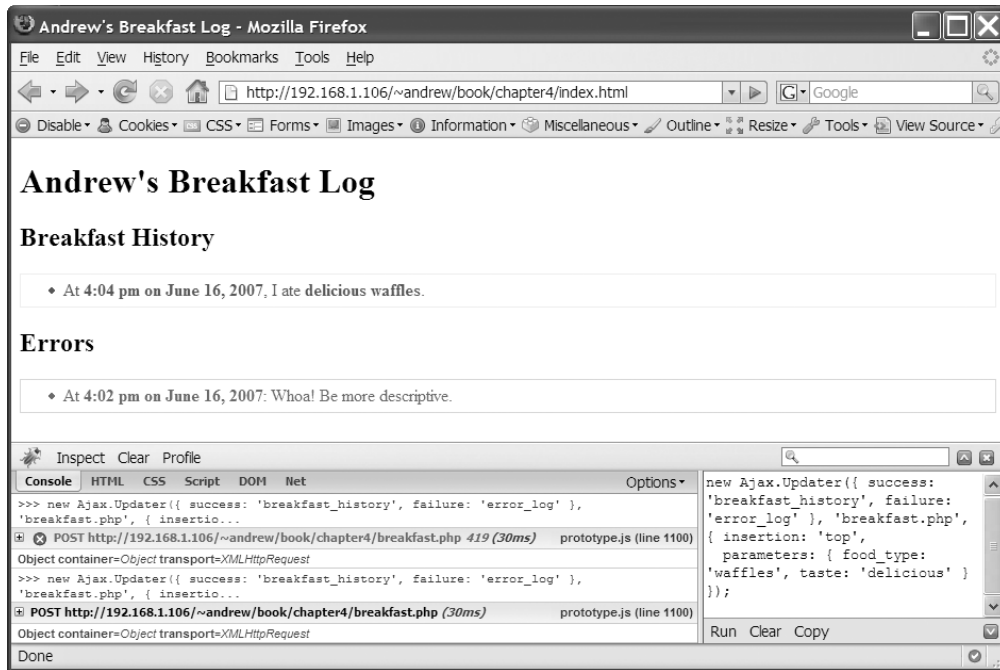


Figure 4-19. Valid entries now look valid.

Example 2: Fantasy Football

It's time to talk about the site we'll be building together over the course of this book. JavaScript has a place on many different kinds of sites, but web *applications* are especially ripe for applying the topics we'll cover in the pages ahead. So let's build a site that people need to *use*, rather than just read. We'll build the sort of site that Ajax can have the greatest impact on.

The App

For the next few chapters, we'll be building a *fantasy football* site. For those who don't know (apologies to my non-American readers), fantasy football is a popular activity among followers of professional American football. Here's fantasy football in a nutshell:

- A group of friends will form a league, “draft” certain real-life players, and earn points each week based on the in-game performances of these players. The members of the league, called “owners,” have duties much like the coaches and managers of real-life football teams: they can make trades, sign free agents, and decide who plays and who sits on the bench (“starters” and “reserves”).
- Each week, a fantasy team will compete against another fantasy team to see who can score more points. (Nearly all pro football games are played on Sunday; nearly all pro teams play every single week.) Owners must decide who to start by predicting which of their players will score the most points. This can vary from week to week based on real-life match-ups, injuries, and other factors.
- A fantasy team earns points whenever one of its starters does something notable in his real-life game. For instance, when a player scores a touchdown in a game, any fantasy football owners who started him that week will earn points. Players also earn points for rushing yardage (advancing the ball on the ground), passing yardage (advancing the ball through the air), and field goals (kicking the ball through the uprights). Scoring systems vary from league to league, but these activities are the ones most often rewarded.
- Fantasy football has been around for several decades, but was far more tedious before computers; owners would have to tabulate their scores manually by reading game results in the newspaper. The Web eliminated this chore, thus causing an explosion in the game’s popularity. The National Football League (NFL), America’s top professional league, even offers free fantasy football on its own web site.

If none of this makes sense to you, don’t worry. All you need to know is that we’re building a web application that will need plenty of the bells and whistles we’ll be covering in the chapters ahead.

The League

Most “reliable sources” state that American football originated in the United Kingdom sometime in the 1800s as an offshoot of rugby. This assertion, while “true,” is breathtakingly dull. Wouldn’t it be much more interesting if American football had, in fact, been conceived of by the nation’s *founding fathers*? Wouldn’t you be fascinated to learn that, during the framing of the Constitution, members of the Congress of the Confederation often resolved disputes by advancing an oblong ball down a grassy field on the outskirts of Philadelphia?

There are no surviving documents that tell us the makeup of the teams, nor the outcomes of the games, but a thorough reading of the US Constitution gives us hints. And so every year a small group of New England football lovers holds reenactments of these games, complete with costumes and pseudonyms, to pay tribute to the unrecognized progenitors of American football. (And you thought Civil War reenactments were crazy.)

This folk tale, even though I just made it up, allows us to proceed with the fantasy football concept without using the names of *actual* football players—thus sparing me the wrath of the NFL Players' Association, which regulates the use of NFL players' names. So it's a win-win scenario. I avoid a lawsuit; you receive a fun history lesson.

The Scoring

Scoring varies from league to league, but leagues tend to award points when a player does something good for his team. Thus, the players with the best individual statistics are often the highest-scoring fantasy players. Some leagues also subtract points when a player makes a mistake in a game—throwing an interception, for instance, or fumbling the ball.

For simplicity's sake, our league will feature six starting slots per team: one quarterback, two running backs, two wide receivers, and one tight end. These are all offensive players that accrue stats in a way that's easy to measure.

Table 4-1 shows the tasks that will earn points in our league.

Table 4-1. *League Scoring Table*

Task	Typical Position	Points
Throwing a touchdown pass	QB (quarterback)	4
Catching a touchdown pass	WR (wide receiver), TE (tight end), RB (running back)	6
Rushing for a touchdown	RB, QB	6
Every 25 passing yards	QB	1
Every 10 rushing yards	RB, QB	1
Every 10 receiving yards	WR, TE, RB	1

The Stats

And now, finally, we come to this chapter's task. The most important page on a fantasy football site is the *box score* page—the page that shows the score for a given match-up.

Our target user, the avid fantasy football owner, keeps a close eye on the box score every Sunday during football season. He needs a page that can do the following things:

1. Show him the score of the game in a way that he can tell whether he's winning or losing at a glance.
2. Show him his roster and his opponent's roster. Alongside each player should be the number of points he's scored *and* a summary of what he's done in his pro game to earn those points. When the owner's score changes, he should be able to tell which of his players just earned points for him.
3. Show him the scores of the other games in his league. Clicking a particular score should take him to the match-up for that game, where he can view the results in greater detail.
4. Keep the score current *without needing a page refresh*. This page will be open all day.

The first three requirements need to be addressed in the interface. We'll come back to those in the next few chapters. But the last one—updating the content without refreshing—is, quite literally, what Ajax was made for.

Let's translate this into technical requirements:

1. We'll need to use Ajax to refresh the game stats without reloading the page. This means we'll also need a URL that, when requested, will return game stats.
2. The client will send out an Ajax request every 30 seconds to ask the server for new stats.
3. The whole site runs on stats, so other pages we build will *also* need this information. So the data should be sent in an abstract format. The client-side code we write will determine how to format the data for display.
4. For the preceding reason, the client needs to know how to interpret the data it receives. Let's have the server return the stats as JSON; it will be easy to parse in JavaScript.
5. The client should figure out when and how to alert the user to score changes. In other words, it's the client's job to compare the new stats to the previous stats to figure out what's different.

Knowing what we need is one thing; putting it all together is another. Naturally, we'll be focusing on the client side, but the client needs to talk to *something* so that we can ensure our code works right.

But we don't have any real stats; our fictional league's season hasn't started yet. So we'll have to fake it.

Mocking

Think of the client and server as humans. The JavaScript we write deals only with the browser; to make an Ajax call, our code has to ask the browser to make an external request and hand over the response when it's ready. Imagine Alice walking into a room and telling Bob to make a phone call for her. Bob calls Carol, has a quick conversation, and then hangs up and tells Alice what Carol said.

But Alice has no *direct* contact with Carol. If Bob wanted to, he could turn around, pick up the phone, *pretend* to dial a number, *pretend* to have a conversation, hang up, and tell Alice whatever he feels like. The whole thing's a ruse, but Alice is none the wiser.

In the real world, this would be dishonest and unwise behavior. In the computer world, it helps us build our app faster. Lying to your code shouldn't make you feel bad.

The code we write will make a request to a certain URL every 30 seconds; it will expect a JSON response that follows a certain format. But it doesn't need to know *how* those stats are retrieved; that's the job of the server.

For testing purposes, we need a stream of data that behaves the way a *real* stream would during game day: games start out with no score, but points are amassed over time as real-life players do good things.

So let's write a script that will generate some *mock stats* for us. This script can be told which stats to report at any given time. It can behave the way real stats would.

We'll be using PHP, but the basic idea is the same for any language. Here's how it will work:

1. We define several classes—one or two for each position. These will represent the players. Each will score at a different rate.
2. We'll have stats running on a 10-minute cycle, after which every score will reset. This may seem like a short cycle—and, indeed, it's much shorter than a real football game—but a shorter cycle only helps us develop more quickly.
3. We'll tell each class how to report its own score based on the current time and the pace of scoring we've set.

We'll write as little code as possible in order to get this done. This won't be a part of the final site, so it doesn't have to be pretty; it just has to work.

The Data

Since PHP 5.2, when the JSON module was first included in a default PHP installation, it has been easy to encode and decode JSON in PHP. The `json_encode` function is part of the core language:

```

$data = array(
    "QB" => "Alexander Hamilton",
    "RB" => "John Jay",
    "WR" => "James Madison"
);

json_encode($data);
//-> '{ "QB": "Alexander Hamilton", "RB": "John Jay", "WR": "James Madison" }'

```

This works with nested arrays as well (arrays that contain arrays):

```

$teams = array(
    "team1" => array(
        "QB" => "Alexander Hamilton",
        "RB" => "John Jay",
        "WR" => "James Madison"
    ),
    "team2" => array(
        "QB" => "George Washington",
        "RB" => "John Adams",
        "WR" => "John Hancock"
    )
);

json_encode($teams);

//-> '{
//->   "team1": {
//->     "QB": "Alexander Hamilton",
//->     "RB": "John Jay",
//->     "WR": "James Madison"
//->   },
//->   "team2": {
//->     "QB": "George Washington",
//->     "RB": "John Adams",
//->     "WR": "John Hancock"
//->   }
//-> }'

```

This is great news. It means we can create a data structure using nested associative arrays (PHP's equivalent to JavaScript's Object type), waiting until the very last step to convert it to JSON.

So let's decide on a structure for the data we'll receive. Hierarchically, it would look something like this:

- Team 1
 - Score
 - Players
 - Yards
 - Touchdowns
 - Score
 - Summary
- Team 2
 - Score
 - Players
 - Yards
 - Touchdowns
 - Score
 - Summary

In pure JavaScript, we would build this with nested object literals. Since we're in PHP, though, we'll use associative arrays.

If you're using a different server-side language, don't worry—JSON libraries exist for practically every commonly used programming language. The concept is the same.

The Code

Let's figure out how to keep track of the time, since that's the most important part. We don't care what the time is in *absolute* terms; we just care about setting milestones every 10 minutes, and then checking how long it has been since the last milestone. Sounds like a job for the modulus operator.

```
// figure out where we are in the 10-minute interval
$time = time() % 600;
```

PHP's `time` function gives us a UNIX timestamp (the number of seconds since January 1, 1970). There are 600 seconds in 10 minutes, so we divide the timestamp value by 600 and take the remainder. This will give us a value between 0 and 599.

First, we grab the timestamp. All we're looking for is a number from 0 to 600 (telling us where we are in the 600-second cycle), so we'll use the modulus operator on a standard UNIX timestamp.

All player classes will need this value, so we'll write a base `Player` class that will define the time instance variable.

```
class Player {
    var $time;
    function Player() {
        global $time;
        $this->time = $time;
    }
}
```

In PHP, we make a constructor by defining a function with the same name as its class. So the `Player` function will get called whenever we declare a new `Player` (or any class that descends from `Player`). All this constructor does is store a local copy of `$time`. (Pulling in `$time` as a global is a little sloppy, but it's quicker.)

Now, by writing position-specific classes that extend `Player`, we can do different things with the time variable in order to report different stats. These classes will have two things in common:

- Each will define a `stats` method that will return the player's stats thus far in the 10-minute cycle.
- The stats will be returned in array form, with fields for yards, touchdowns, fantasy points scored, and a text summary of the player's performance. This structure will be converted to JSON when it's sent over the pipeline.

A quarterback would slowly accrue passing yards over a game—with the occasional touchdown pass in between. Since we're compressing a whole game's statistics into a 10-minute period, we should set a faster pace.

```
// QB throws for 10 yards every 30 seconds
// and a touchdown every 4 minutes.
class QB extends Player {
    function stats() {
        $yards = floor($this->time / 30) * 10;
        $tds   = floor($this->time / 240);
```

```

    return array(
        "yards" => $yards,
        "TD"    => $tds,
        "points" => floor($yards / 25) + (4 * $tds),
        "summary" => $yards . " yards passing, " . $tds . " TD"
    );
}
}

```

We're extending the `Player` class, so we get its constructor for free. All we have to define, then, is the `stats` method. To get a score from this class, you need only declare a new instance and call this method.

```

$time = 430; // let's say
$qb = new QB();
$qb->score ();

//-> array(
//->  "yards"    => 140
//->  "TD"       => 1,
//->  "points"   => 9,
//->  "summary"  => "140 yards passing, 1 TD"
//-> )

```

Now we'll do the same for the running back and wide receiver. But, since a team starts two running backs and two wide receivers (as opposed to starting one quarterback), we should make two different classes for each of these positions. That way, by mixing and matching which combinations start for which team, we can introduce some variation in the scoring.

```

// RB1 runs for 5 yards every 30 seconds and scores at minute #6.
class RB1 extends Player {
    function stats() {
        $yards = floor($this->time / 30) * 5;
        $tds   = floor($this->time / 360);
    }
}

```

```

return array(
    "yards"    => $yards,
    "TD"       => $tds,
    "points"   => floor($yards / 10) + (6 * $tds),
    "summary"  => $yards . " yards rushing, " . $tds . " TD"
);
}
}

```

// RB2 runs for 5 yards every 40 seconds and does not score.

```

class RB2 extends Player {
    function stats() {
        $yards = floor($this->time / 40) * 5;

        return array(
            "yards"    => $yards,
            "TD"       => 0,
            "points"   => floor($yards / 10),
            "summary"  => $yards . " yards rushing, 0 TD"
        );
    }
}

```

// WR makes one catch every minute for 15 yds and scores at minute #4.

```

class WR1 extends Player {
    function stats() {
        $yards = floor($this->time / 60) * 15;
        $tds   = $this->time > 240 ? 1 : 0;

        return array(
            "yards"    => $yards,
            "TD"       => $tds,
            "points"   => floor($yards / 10) + (6 * $tds),
            "summary"  => $yards . " yards receiving, " . $tds . " TD"
        );
    }
}

```

```
// WR makes one catch every 2 minutes for 25 yds and does not score.
class WR2 extends Player {
  function stats() {
    $yards = floor($this->time / 120) * 25;

    return array(
      "yards" => $yards,
      "TD"    => 0,
      "points" => floor($yards / 10),
      "summary" => $yards . " yards receiving, 0 TD"
    );
  }
}
```

These classes all return data in the same format. They only differ in the “script” they follow—the way they turn that original \$time value into a point total.

Each team will start only one tight end, so we needn’t bother with more than one “version” of tight end.

```
// TE makes one catch at minute #8 for a 20-yard TD.
class TE extends Player {
  function stats() {
    $yards = $this->time > 480 ? 20 : 0;
    $tds   = $this->time > 480 ? 1 : 0;

    return array(
      "yards" => $yards,
      "TD"    => $tds,
      "points" => floor($yards / 10) + (6 * $tds),
      "summary" => $yards . " yards receiving, " . $tds . " TD"
    );
  }
}
```

There’s only one thing left to do: organize these players into teams. At the bottom of scores.php, we’ll add the code to do this and output to JSON.

```
// Adds a player's score to a running total; used to
// compute a team's total score
function score_sum($a, $b) {
  $a += $b["points"];
  return $a;
}
```

```
$qb = new QB();
$rb1 = new RB1();
$rb2 = new RB2();
$wr1 = new WR1();
$wr2 = new WR2();
$te = new TE();

$team1 = array();

// team 1 will score more points, so we give it
// the better "versions" of RB and WR
$team1["players"] = array(
    "QB" => $qb->stats(),
    "RB1" => $rb1->stats(),
    "RB2" => $rb1->stats(),
    "WR1" => $wr1->stats(),
    "WR2" => $wr1->stats(),
    "TE" => $te->stats()
);

// take the sum of all the players' scores
$team1["points"] = array_reduce($team1["players"], "score_sum");

$team2 = array();

// team 2 will score fewer points, so we give it
// both "versions" of RB and WR
$team2["players"] = array(
    "QB" => $qb->stats(),
    "RB1" => $rb1->stats(),
    "RB2" => $rb2->stats(),
    "WR1" => $wr1->stats(),
    "WR2" => $wr2->stats(),
    "TE" => $te->stats()
);

// take the sum of all the players' scores
$team2["score"] = array_reduce($team2["players"], "score_sum");

// deliver it in one large JSON chunk
echo json_encode(array("team_1" => $team1, "team_2" => $team2));
```

To paraphrase Blaise Pascal: I apologize for writing a long script, but I lack the time to write a short one. We could have taken the time to write more elegant code, but why? This script doesn't need to be maintainable; it just needs to work. And football season is fast approaching. Better to take extra care with the code that the *general public* will see.

Testing It Out

It will be easy to see whether our script works—we need only open it in a browser. Fire up Firefox and type the URL to your `scores.php` file.

If all goes well, you should see some JSON on your screen (see Figure 4-20).

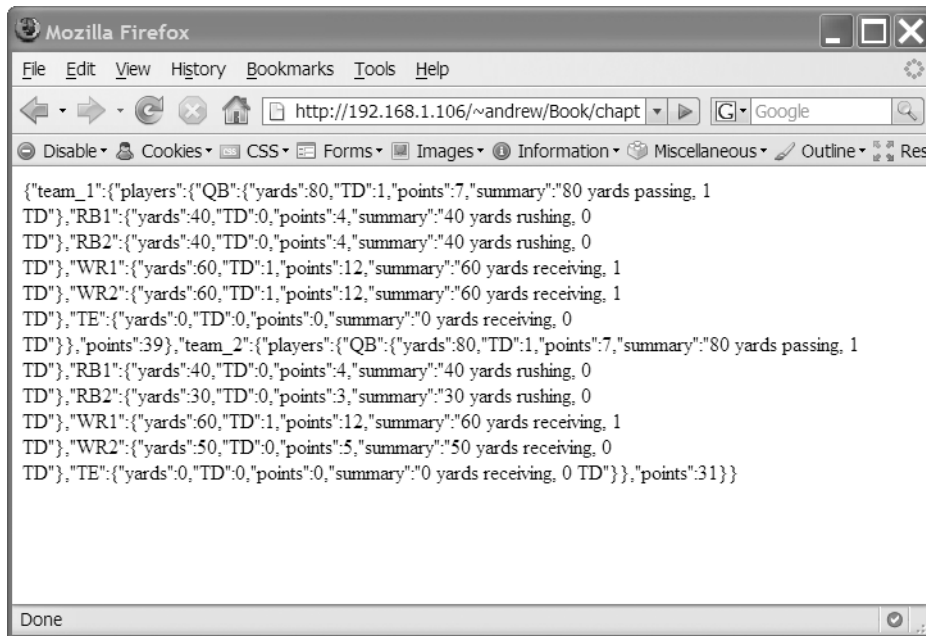


Figure 4-20. The raw data generated by our script

The numbers on your screen will vary from those in Figure 4-20. Because they run off a 10-minute cycle, the last digit of your system time (in minutes) is the factor—the closer it is to 0, the closer the scores will be to 0. Reload your page in 30 seconds and some of the scores will increment—and will continue to increment until that minute hand hits another multiple of 10, at which time the scores will all go back to 0.

We have spent a lot of time on `scores.php`, but it will save us much more time later on. We've just written a simulation of nearly all the data our site needs from the outside world.

Making an Ajax Call

Finally, we come to the Ajax aspect of this example. Create a blank `index.html` file in the same directory as your `scores.php` file. It shouldn't be completely empty—make sure it loads `prototype.js`—but it doesn't need any content. From here we can use the Firebug shell to call our PHP script and look at the response.

Open `index.html` in a browser, and then open the Firebug console and type the following:

```
var request = new Ajax.Request("scores.php");
```

Firebug logs all the details about the Ajax request, as shown in Figure 4-21.



Figure 4-21. Our Ajax request in the Firebug console

Expand this line, and then click the Response tab (see Figure 4-22).

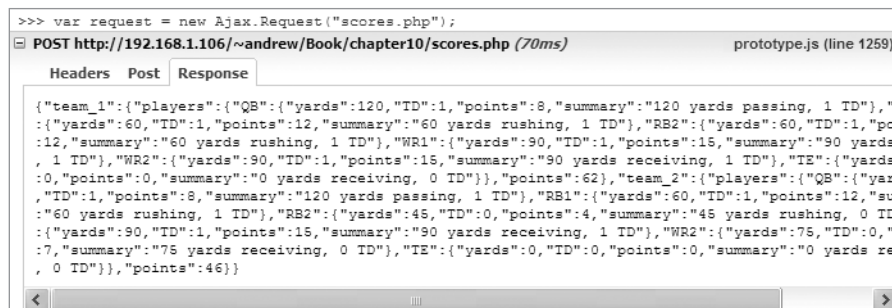


Figure 4-22. The same data we saw in Figure 4-20

There's our JSON, brackets and everything. Typing `request.responseText` into the Firebug console will give you the response in string form.

We can do better than that, though. Go back to the request details, and then switch to the Headers tab. There are two sets of headers—*request* headers and *response* headers—corresponding to the headers we sent out and the headers we got back, respectively. The response headers should tell you that our JSON data was served with a Content-type of `text/html`.

It's *not* HTML, though; PHP just serves up everything as HTML by default. We can tell our script to override this default. The de facto Content-type for JSON is `application/json`, so let's use that.

Go back to `scores.php` (last time, I promise) and insert the following bold line near the bottom:

```
// deliver it in one large JSON chunk
header("Content-type: application/json");
echo json_encode(array("team_1" => $team1, "team_2" => $team2));
```

This call to the header function will set the proper Content-type header for the response.

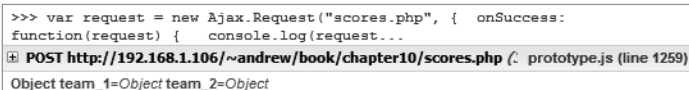
Caution You must call the header function before any output has been placed in the response. This includes anything printed or echoed, plus anything that occurs in your script before the PHP start tag (`<?php`). Even *line breaks* count as output.

Save your changes, and then go to the Firebug console. Press the up arrow key to recall the last statement you typed, and then press Enter. Inspect the details of this request and you'll notice that the Content-type has changed to `application/json`.

Why did we bother with this? It's not just a compulsion of mine; I promise. When Prototype's `Ajax.Request` sees the `application/json` content type, it knows what sort of response to expect. It unserializes the JSON string *automatically*, creating a new property on the response. To prove it, we'll try one more statement in the Firebug console. (You may want to switch to multiline mode for this one.)

```
var request = new Ajax.Request("scores.php", {
  onSuccess: function(request) {
    console.log(request.responseJSON);
  }
});
```

Run this statement; then watch a miracle happen in your console (see Figure 4-23).



```
>>> var request = new Ajax.Request("scores.php", { onSuccess:
function(request) { console.log(request.responseJSON);
}
});
POST http://192.168.1.106/~andrew/book/chapter10/scores.php (line 1259)
Object team_1=Object team_2=Object
```

Figure 4-23. Our data. But it's no longer raw.

Egad! That looks like our data. Click the bottom line to inspect the object—the JSON response has been converted to `Object` form automatically.

Let's recap what we've learned about the different Ajax response formats:

- All requests, no matter what the Content-type of the response, bear a `responseText` property that holds a string representation of the response.
- Requests that carry an XML Content-type bear a `responseXML` property that holds a DOM representation of the response.
- Prototype extends this pattern to JSON responses. Requests that carry a JSON Content-type bear a `responseJSON` property that holds an `Object` representation of the response.

The `responseJSON` property, while nonstandard, is the natural extension of an existing convention. It simplifies the very common pattern of transporting a data structure from server to client, converting the payload into the data type it's meant to be.

Summary

The code you've written in this chapter demonstrates the flexible design of Prototype's Ajax classes—simple on the surface, but robust on the inside. As the examples went from simple to complex, the amount of code you wrote increased in modest proportion.

You typed all your code into Firebug because you're just starting out—as you learn about other aspects of Prototype, we'll mix them in with what you already know, thus pushing the examples closer and closer to real-world situations. The next chapter, all about events, gives us a big push in that direction.

