PART

Advanced Topics

CHAPTER 10
Web Services and Beyond

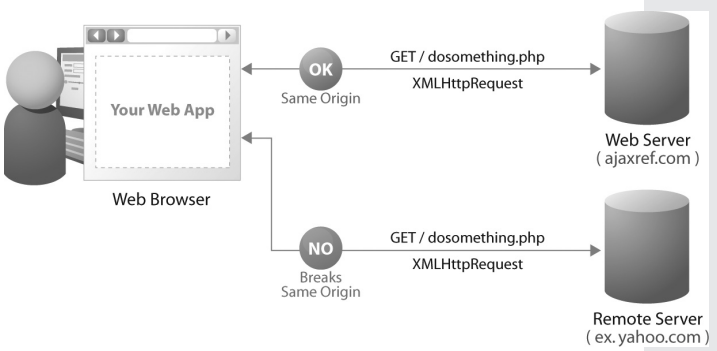
CHAPTER 10

Web Services and Beyond

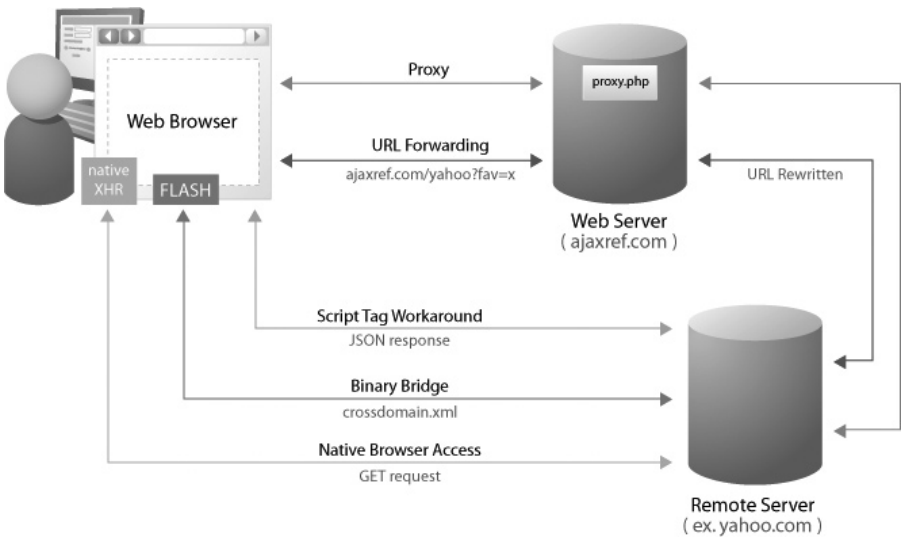
Ajax and related ideas are changing at a furious pace. In this chapter we present but a brief overview of a few of the most important areas of change in the world of Ajax, including the use of remote data and application in the form of Web Services, a push-style communication pattern generally dubbed Comet, and the final missing piece so that Web applications can compete with desktop apps: offline storage and operation. Given the tremendous rate of innovation in each of these areas, our aim is to present an overview of the idea, a discussion of some of the ramifications and concerns surrounding it, and a representative example or two with a bit less emphasis on syntax specifics than general approach. That's not to say that we won't provide working examples—there are plenty to be found here—but compared to those presented in earlier chapters these are more likely to break as APIs outside of our control change. As such, we encourage readers to visit the book support site for the latest info in case they encounter problems. So, with warning in hand, let us begin our exploration of the bleeding edges of Ajax.

Ajax and Web Services

Ajax and Web Services are often mentioned in the same breath, which is quite interesting considering that as of yet they really do not work well together. As we have seen throughout the book, at this moment in time (late 2007), the same origin policy restricts cross-domain requests that would be mandatory in order to use a Web Service directly from client-side JavaScript. For example, if you desired to build a Web page and hosted it on your server (example.com) and call a Web Service on (google.com), you could not do so directly using an XHR.



However, there are ways around this limitation as shown in the diagram here and summarized in Table 10-1.



Approach	Description	Comments
Proxy	Calls a script on the server of delivery (within same origin) that calls remote Web Service on your behalf and passes the result back.	Avoids same origin issue. Puts burden on your server to forward requests. May provide a proxy that can be exploited.
URL forwarding	A variation of the previous method. Calls a URL on the server (within same origin), which acts as a proxy redirect that pipes data transparently to a remote resource and back. Usually performed using a server extension like mod_rewrite.	Avoids same origin issue. Puts burden on your server to forward requests. May provide a proxy that can be exploited.
Script Tag Workaround	Makes call to remote service using a `<script>` tag that returns a wrapped JSON response invoking a function in the hosting page.	Not restricted by same origin. Script transport not as flexible as XHR. Script responses and JSON responses shown to have some security concerns these might be mitigated with browser changes or the iframe solution discussed in Chapter 7.

Table 10-1 Summary of Web Service via Ajax Approaches

Approach	Description	Comments
Binary Bridge	Uses Flash or Java applet to make a connection to another domain. In the case of Flash this relies on a trust-relationship defined on the target server specified in a crossdomain.xml file.	Relies on binary that may not be installed. Piping between JavaScript and binary may be problematic. Requires configuration of remote resource to allow for access. May allow for other communication methods (for example, sockets) and binary data formats.
Native Browser Access	In emerging browsers like Firefox 3 you should be able to make a basic GET request with an XHR outside of origin as long as there is a trust relationship defined (similar to binary bridge solution).	Uses native XHR. Requires configuration of remote resource to allow for access. Not widely implemented as of yet.

TABLE 10-1 Summary of Web Service via Ajax Approaches (*continued*)

Server Proxy Solution

The basic idea of a server proxy solution is to submit a request to a server-side program via an Ajax call, and then that program either passes the request on or triggers a new request to a Web Service on your behalf. The packet returned from the Web Service can either be modified before being passed back or just passed on in a raw form. While it may sound involved to set up, it isn't terribly difficult. As an example, the rough algorithm for a transparent forwarding proxy is something like:

```
define URL you want to call
read the data from the Ajax request
form full query to Web service in question
issue request and save back results
begin response by printing headers
if status of service call != 200
   pass back error message
else
   pass back results
```

As a demonstration, we build a proxy to call the Flickr Web Service to list out images that match a provided keyword. Flickr provides a simple API to do this using a RESTful interface where you can issue simple GET or POST requests to perform actions. Flickr currently has a primary end point URL of:

```
http://api.flickr.com/services/rest/
```

This is where you would send your Web Service requests. All calls to the Flickr API take a parameter `method`, which is the calling method you are interested in; for example, flickr.photos.search to search for photos. You are also required to pass a parameter `api_key`,

which is set to a unique value issued to developers to allow them to make a call. You should register for your own key (www.flickr.com/services/api/) to run demos, as we will not provide a working one in print. Expect that many of the other useful Web Services will require you to register for access as well and use an API key to limit access and abuse. Finally, an optional `format` parameter may be used to indicate what format you would like your reply in:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key=
XXXXXX-FAKE-API-KEY-GET-YOUR-OWN-XXXXXX
```

Besides these basic parameters, you would call the service with a variety of parameters indicating the types of images you are looking for based upon user ID, tag, description, and so on. For example:

```
http://api.flickr.com/services/rest/?method=flickr.photos.search&text=
schnauzer&content_type=1&per_page=10&safe_search=1&api_key=
XXXXXX-FAKE-API-KEY-GET-YOUR-OWN-XXXXXX
```

would perform a safe search for images with a text description containing the word "schnauzer" and then return only images (`content_type`), with ten per page. We'll avoid getting too specific about the API here since it is bound to change. Instead, we point you directly to the online docs since our goal here is solely to understand the general process of using a Web Service with a proxy.

After issuing the request, the Flickr service would respond with some packet like:

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
  Payload here
</rsp>
```

If everything worked right, the contents of the packet would contain a variety of tags depending on what method we invoked. If the request didn't work, we would get a packet response like so:

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="fail">
      <err code="[error-code]" msg="[error-message]" />
</rsp>
```

Here is an actual response for the earlier example query for "schnauzer" pictures, limited to three results.

```
<rsp stat="ok">
 <photos page="1" pages="5993" perpage="3" total="17978">
  <photo id="1297027770" owner="8644851@N05" secret="e7b3330a61" server="1258"
farm="2" title=""Brusca"" ispublic="1" isfriend="0" isfamily="0"/>
  <photo id="1296140191" owner="29807756@N00" secret="a117e20762" server="1077"
farm="2" title="Billy the Kid" ispublic="1" isfriend="0" isfamily="0"/>
  <photo id="1296129605" owner="29807756@N00" secret="c94aa225bf" server="1438"
farm="2" title="Make this move..." ispublic="1" isfriend="0" isfamily="0"/>
</photos>
</rsp>
```

With this process in mind, we see building a simple server proxy is quite easy. For example, quickly read the following PHP code:

```
<?php
header("Cache-Control: no-cache");
header("Pragma: no-cache");
header("Content-Type: text/xml");
$query = $_GET["query"];
$url =  "http://api.flickr.com/services/rest/?method=flickr.photos.search&api_
key=XXXXXXX-FAKE-API-KEY-GET-YOUR-OWN-XXXXX&safe_search=1&per_page=10&content_
type=1&text=$query";
$result = file_get_contents($url);
/* Check response status */
list($version,$status,$msg) = explode(' ',$http_response_header[0], 3);
if ($status != 200)
 echo "Your call to the web service returned an error status of $status.";
else
 echo $result;
?>
```

We see that it takes a value in query and forms the URL to call, then it gets the result and decides whether to pass the packet or send an error message.

To fully develop the example on the client side, we build a simple form to collect the query string in question and then send it off to the proxy program. You'll note that we make sure to set a status indicator here as the request might take a while.


```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Chapter 10 : Flickr Web Service Search using Proxy</title>
<link rel="stylesheet" href="http://ajaxref.com/ch10/global.css" type="text/css"
media="screen" />
<script src="http://ajaxref.com/ch10/ajaxtcr.js" type="text/javascript"></script>
<script type="text/javascript">
function search(searchterm)
{
 if (searchterm == "")
   {
    alert("You must enter a search term");
    return;
   }
 var url =  "http://ajaxref.com/ch10/proxyflickr.php";
 var payload = "query=" + searchterm;
 var options = {method:"GET",
                payload:payload,
                onSuccess: handleResponse,
               statusIndicator : { progress :
{type: "text", text: "Searching...", target: "results"  }}};
 AjaxTCR.comm.sendRequest(url, options);
}
function handleResponse(response)
```

```
{
  var resultsDiv = $id("results");
  resultsDiv.innerHTML = "";

  var images = response.responseXML.getElementsByTagName("photo");
  for (var i=0;i<images.length;i++)
    {
     var image = images[i];
     resultsDiv.innerHTML += "<b>" + image.getAttribute("title") + "</b><br />";
     resultsDiv.innerHTML += "<img src='http://farm" + image.getAttribute("farm")
+ ".static.flickr.com/" + image.getAttribute("server") + "/" + image.
getAttribute("id") + "_" + image.getAttribute("secret") + "_m.jpg' /><br /><br />";
    }
}

window.onload =  function () {
  $id("requestbutton").onclick = function(){search($id("query").value);};
  $id("requestForm").onsubmit = function() {return false;}
};
</script>
</head>
<body>
<div class="content">
<h1>Flickr Search: Server Proxy Version</h1><br />
<form id="requestForm" method="GET" action=
"http://ajaxref.com/ch10/proxyflickr.php" name="requestForm" >
<label>Search Term:
   <input type="text" name="query" id="query" id="query" value="Schnauzer"
autocomplete="off" size="30" />
</label>
<input type="submit" id="requestbutton" value="Search" />
</form>
</div>
<br /><br />
<div id="results" class="results"></div>
</body>
</html>
```

The result of the previous example is shown in Figure 10-1. You can run this for yourself using the demo at http://ajaxref.com/ch10/proxyflickr.html.

Data Differences

The proxy solution shouldn't really care what the end service point returns; it just pipes it all back for your script to consume—but it doesn't have to. For example, if a Web Service returned XML and we needed to consume it as JSON, we could rewrite the content in the server proxy to deal with that. Here's the outline of the kind of code that would do that for our example:

```
<?php
require_once('XML2JSON.php');
header("Cache-Control: no-cache");
header("Pragma: no-cache");
header("Content-Type: application/json");
```

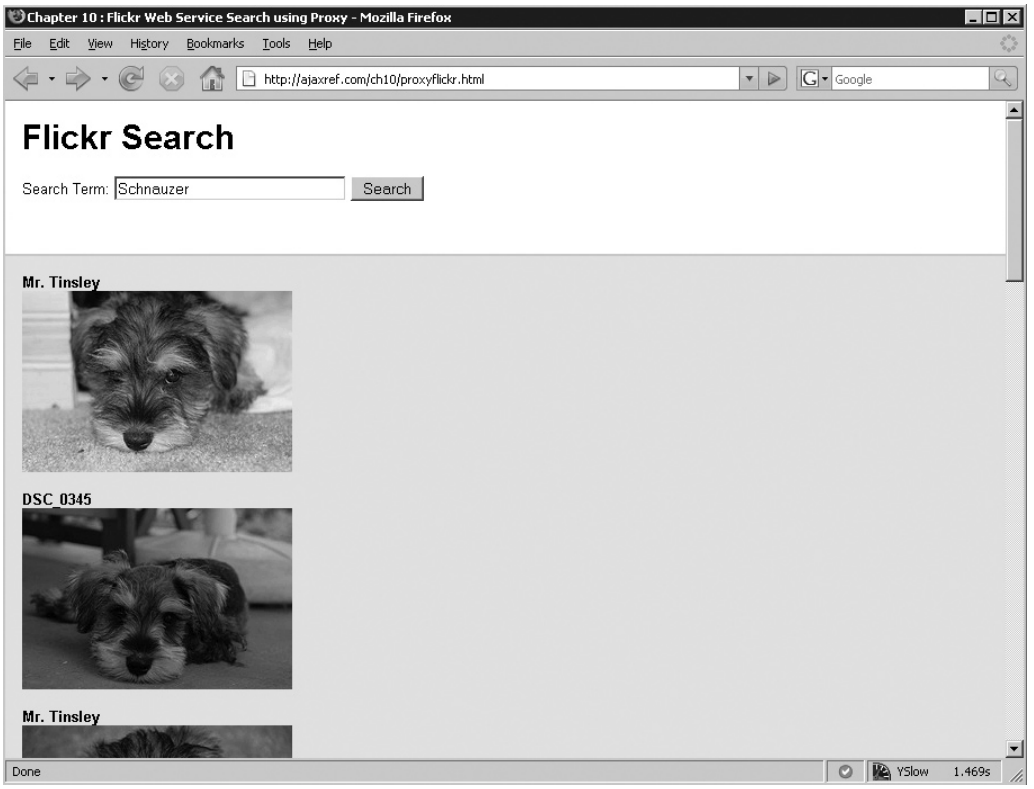


FIGURE 10-1 Searching for pictures via a Web Service

```
$query = $_GET["query"];
$url =  "http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key=
dc11b3d4586f0d925629fe2ed5cf250a&safe_search=1&per_page=
10&content_type=1&text=$query";
$result = file_get_contents($url);
/* Check response status */
list($version,$status,$msg) = explode(' ',$http_response_header[0], 3);

if ($status != 200)
 echo "Your call to the web service returned an error status of $status.";
else
 {
  /* take XML string and make DOM tree */
  $domtree = new DOMDocument();
  $domtree->loadXML($result);
  /* convert from XML to JSON */
  $transform = new XML2JSON();
  $result = $transform->convertToJSON($domtree);
  print $result;
 }
?>
```

The details of the conversion are not terribly illuminating: you don't have to pass results raw to the client; you are free to filter or even combine them with other data. We'll see that idea later in the chapter when we discuss mash-ups.

Many Web Services provide output options so you do not have to convert their data format to the one you prefer. The Flickr API provides multiple output formats that can be requested by setting the format. We can pass the parameter (`format=json`) and get back the same type of information as was found in the XML packet but in a wrapped JSON format, like so:

```
jsonFlickrApi({"photos":
      {"page":1,
       "pages":4495,
       "perpage":3,
       "total":"17978",
       "photo":[{"id":"1296140191", "owner":"29807756@N00",
"secret":"a117e20762", "server":"1077", "farm":2, "title":"Billy the Kid",
"ispublic":1, "isfriend":0, "isfamily":0},
                {"id":"1296129605", "owner":"29807756@N00",
"secret":"c94aa225bf", "server":"1438", "farm":2, "title":"Make this move...",
"ispublic":1, "isfriend":0, "isfamily":0},
                {"id":"1296081377", "owner":"29807756@N00",
"secret":"2e0d71c879", "server":"1413", "farm":2, "title":"Clueless",
"ispublic":1, "isfriend":0, "isfamily":0},
]},
        "stat":"ok"}
)
```

Note the call to the function `jsonFlickrApi()`, which is what they would want you to name a default callback function. You can change that using the parameter `jsoncallback`, so we might set something like `jsoncallback=formatOutput` in our request. You can also eliminate the callback and just pass back the raw JSON packet using the parameter `nojsoncallback=1` in the query string. Our emphasis on this other data format will become clear in a second when we discuss bypassing the proxy approach all together.

URL Forwarding Scheme

While the previous approach works reasonably well, we do have to write a program to handle the request. It might be convenient instead to call a particular URL and have it automatically forward our requests. For example, we might employ `mod_proxy` and `mod_rewrite` for Apache to enable such functionality. Setting a rule in Apache's config file like the one below performs a core piece of the desired action.

```
ProxyPass /flikrprox http://api.flickr.com/services/rest/
```

Here we indicated that a request on our server to `/flickrprox` will pass along the request to the remote server. From our Ajax application we would then create a URL like:

```
http://ajaxref.com/flikrprox/?method=flickr.photos.search&api_key=XXXX-GET-
YOUR-OWN-KEY-XXXX&safe_search=1&per_page=10&content_type=1&text=Schnauzer
```

As we show here:

```
var url =  "http://ajaxref.com/flickprox";
var flickrMethod = "flickr.photos.search";
var flickrAPIKey = "XXXX-GET-YOUR-OWN-KEY-XXXX";
var payload="?method="+flickrMethod+"&api_key"+flickrAPIKey+
"&safe_search=1&per_page=10&content_type=1&";
url+= "text=" + searchterm;

var options = {method:"GET",
               payload:payload,
               onSuccess: handleResponse,
              statusIndicator : { progress : {type: "text", text:
"Searching...", target: "results"  }}};
 AjaxTCR.comm.sendRequest(url, options);
```

and it passes it along to the Flickr site and returns our response packet back to us.

It should be obvious that this approach leaves the URL redirection proxy open to being abused, but only for that specific site, which is not as bad as leaving it wide open for anything. We also note that the use of the proxy is not limited to just our API key, which will also be exposed in the JavaScript and is likely not appropriate to disclose. A better solution would be to create a rewrite rule on the server to hide some of these details in the rewrite and then pass on the request in the proxy fashion. Here is a snippet from an apache.config file that would do this for our example:

```
RewriteRule ^/flickrprox http://api.flickr.com/services/rest/?method=
flickr.photos.search&api_key=xxx-YOUR-API-KEY-HERE-xxx&safe_search=
1&per_page=10&content_type=1 [QSA,P]

ProxyRequests Off
<Proxy *>
Order deny,allow
Allow from all
</Proxy>
ProxyPass /flickrprox http://api.flickr.com/services/rest
```

With this rule in place we do not have to expose many details in the source as seen here. You could, of course, rewrite this only to add in the API key in the server-rule, but we show the example with many variables so you can see that you can perform quite complex rewrites if you like.

NOTE *URL rewriting and proxying on a Web server can involve some seriously arcane Web knowledge. We have only skimmed the surface of this topic to show you the possibility of using the approach. If this approach seems appealing to you, spend some time getting to know* `mod_rewrite` *or your server's equivalent before approaching the kind of example we presented. It will save you significant frustration.*

A working version of the URL rewrite-proxy approach can be found at http://ajaxref.com/ch10/urlrewriteproxyflickr.html and is shown in action in Figure 10-2. Notice in the figure that the network trace clearly shows you do not have a chance on the client side to see the URL rewriting with the API key in it, and thus the secret is protected.

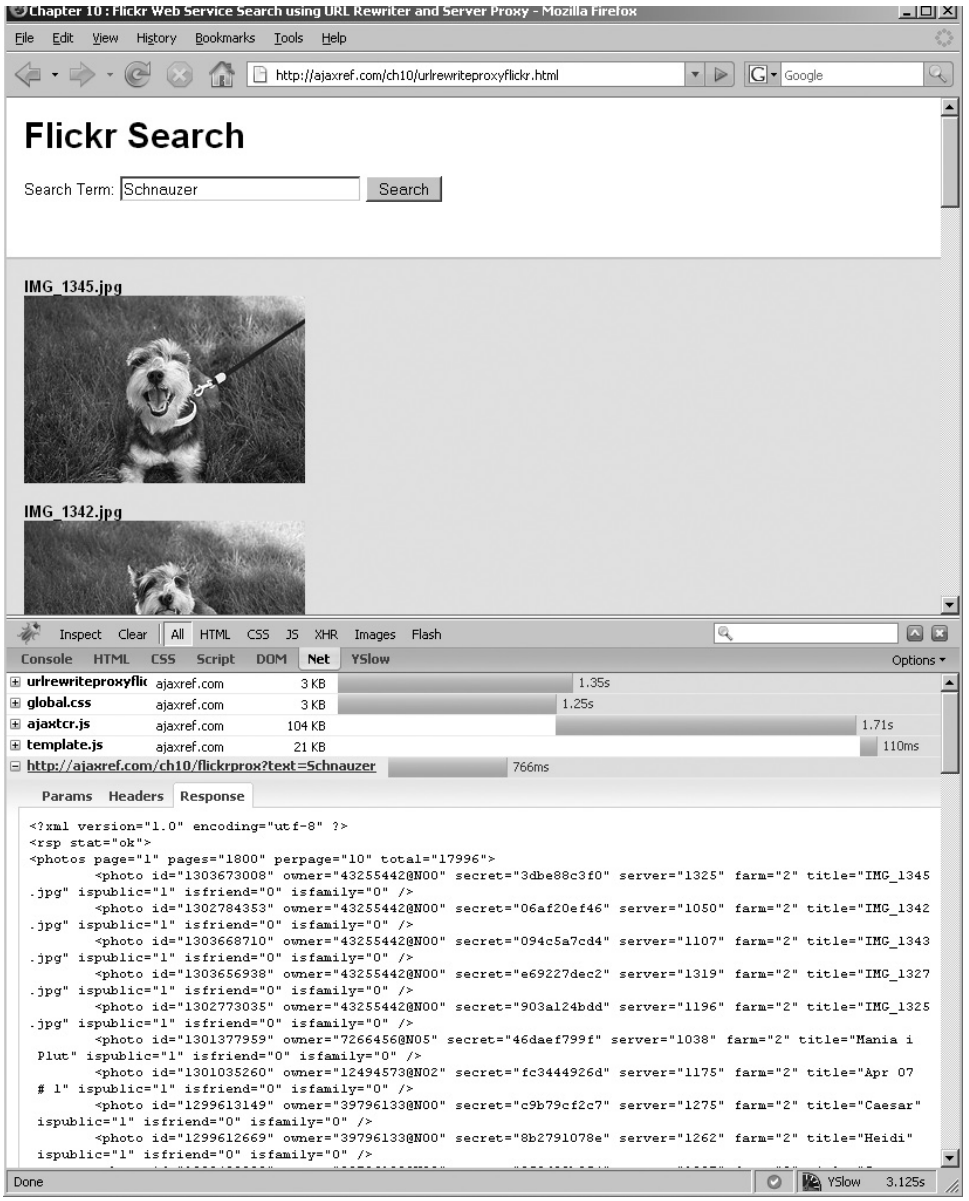


FIGURE 10-2 URL forwarding proxy to enable Ajax Web Service calls in action

Using the <script> Tag

In both of the previous cases we are relying on the server to help us out, but we should hope for a more direct path to the Web Service. Given the same origin policy and current restrictions for cross-domain XHR requests found in browsers circa late 2007, we are forced to look for alternate transport. The `<script>` tag transport that has been discussed since

Chapter 2 is our best candidate. Response-wise we will of course expect JavaScript—usually a wrapped JSON packet or some raw JavaScript—to execute. We continue with Flickr as it provides a remote `<script>` call interface as well.

In the case of Flickr, we saw that their JSON packet is by default wrapped with a function call like so:

```
jsonFlickrApi(   {JSON object}  )
```

Here the JSON object is a representation of the `<rsp>` element found in the typical RESTful response. Recall that we can change the callback to our own function name by passing a `jsoncallback` parameter (`jsoncallback=handleResponse`). To execute our `<script>` tag Web Service approach, we will need to set all the parameters to the service ourselves, so we make a simple object to hold all of them.

```
var flickrConfig = {
 method : "flickr.photos.search",
 api_key : "dc11b-FAKE-KEY-HERE0--a",
 safe_search : 1,
 per_page : 10,
 content_type : 1,
 format : "json",
 jsoncallback : "handleResponse"
};
```

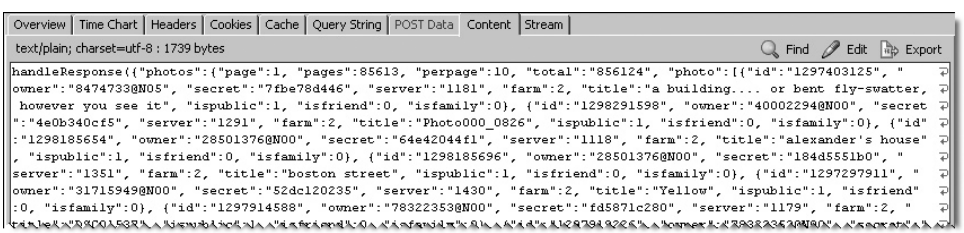
Now we set up our payload to contain all the items as well as the search term using our handy `AjaxTCR.data.serializeObject()` method:

```
var payload = "text=" + searchterm;
 payload = AjaxTCR.data.serializeObject(payload,flickrConfig,"application/
x-www-form-urlencoded");
```

Given that since Chapter 9 we've supported other transports in our library, we just indicate we want to use a `<script>` tag instead of an XHR when making our request:

```
var url =  "http://api.flickr.com/services/rest/";
var options = {method:"GET",
               payload:payload,
               transport: "script",
               statusIndicator : { progress : {type: "text", text:
"Searching...", target: "progress"  }}};
 AjaxTCR.comm.sendRequest(url, options);
```

We don't specify the callback, of course, since the payload will contain it. Now we should receive a response like so:



As you can see, this response performs its own callback, so to speak, by invoking `handleResponse()`. This function then takes the passed object and creates the `<img>` tags to fetch the images of interest from Flickr.

```
function handleResponse(response)
{
  var resultsDiv = $id("results");
  resultsDiv.innerHTML = "";
  if (response.stat == "ok")
   {
    var images = response.photos.photo;
    for (var i=0;i<images.length;i++)
    {
     var image = images[i];
     resultsDiv.innerHTML += "<b>" + image.title + "</b><br />";
     resultsDiv.innerHTML += "<img src='http://farm" + image.farm +
".static.flickr.com/" + image.server + "/" + image.id + "_" + image.secret
+ "_m.jpg' /><br /><br />";
    }
   }
  else
    resultsDiv.innerHTML = "<h2>An error has occurred</h2>";
}
```

The complete code is shown next and demonstrated in Figure 10-3. A live example can be found at http://ajaxref.com/ch10/scriptflickr.html.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Chapter 10 : Flickr Web Service Search using Script-JSON</title>
<link rel="stylesheet" href="http://ajaxref.com/ch10/global.css" type="text/css"
media="screen" />
<script src="http://ajaxref.com/ch10/ajaxtcr.js" type="text/javascript"></script>
<script type="text/javascript">
var flickrConfig = {
 method : "flickr.photos.search",
 api_key : "dc-FAKE-KEY-HERE-GET-YOURS-250a",
 safe_search : 1,
 per_page : 10,
 content_type : 1,
 format : "json",
 jsoncallback : "handleResponse"
};

function search(searchterm)
{
 if (searchterm == "")
   {
    alert("You must enter a search term");
    return;
```

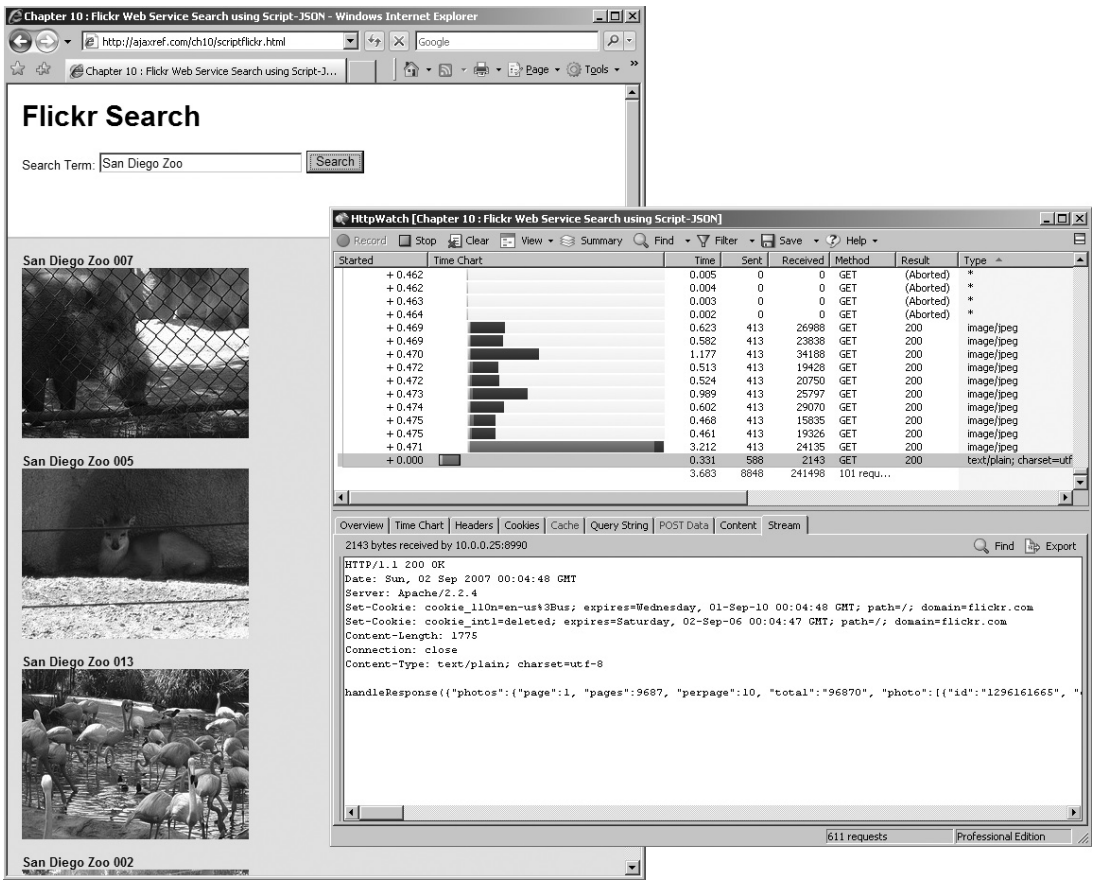


FIGURE 10-3 Using direct response from Flickr Web Service via `<script>` call

```
   }
 var url =  "http://api.flickr.com/services/rest/";
 var payload = "text=" + searchterm;
 payload = AjaxTCR.data.serializeObject(payload,flickrConfig,
"application/x-www-form-urlencoded");
 var options = {method:"GET",
                payload:payload,
                transport: "script",
                statusIndicator : { progress :
{type: "text", text: "Searching...", target: "progress"  }}};
 AjaxTCR.comm.sendRequest(url, options);
}
function handleResponse(response)
{
  var resultsDiv = $id("results");
  resultsDiv.innerHTML = "";
```

```
  if (response.stat == "ok")
   {
    var images = response.photos.photo;
    for (var i=0;i<images.length;i++)
    {
     var image = images[i];
     resultsDiv.innerHTML += "<b>" + image.title + "</b><br />";
     resultsDiv.innerHTML += "<img src='http://farm" + image.farm +
".static.flickr.com/" + image.server + "/" + image.id + "_" + image.secret +
"_m.jpg' /><br /><br />";
    }
   }
  else
    resultsDiv.innerHTML = "<h2>An error has occurred</h2>";
}
window.onload =  function () {
  $id("requestbutton").onclick = function(){search($id('query').value);};
  $id("requestForm").onsubmit = function() {return false;}
};
</script>
</head>
<body>
<div class="content">
<h1>Flickr Search: Script/JSON Version</h1><br />
<form id="requestForm" method="GET" action=
"http://ajaxref.com/ch10/proxyflickr.php" name="requestForm" >
<label>Search Term:
   <input type="text" name="query" id="query" id="query" value="Schnauzer"
autocomplete="off" size="30" />
</label>
<input type="submit" id="requestbutton" value="Search" />
</form>
</div>
<br /><br /><div id="progress"></div>
<div id="results" class="results"></div>
</body>
</html>
```

While the `<script>` tag does let us break the same origin policy, we should do so with caution. As demonstrated in Chapter 7, untrustworthy sites can introduce problems even with JSON payload responses. There is a somewhat inelegant solution using a number of iframes often dubbed "subspace" that can be employed, but you will have to be quite careful with testing to ensure a robust connection. We point readers back to the security discussion (Chapter 7) for more information, but for now, since we have found one client-side focused way to break the SOP, you might wonder if there are other approaches. But of course!

Flash Cross Domain Bridge

We saw that the `<script>` tag can break the same origin, but it turns out there is something else that we could use that might be a bit more flexible to perform this action: Flash. Generally people tend to think of Flash for animation, video, and various rich applications. However, if you dig deeper into Flash you come to realize that it has a rich development

environment complete with a number of useful communication features. For example, in ActionScript you can load a document from a remote resource very quickly.

```
var myXML = new XML();
myXML.load(url);  /* url contains the address we want to load */
```

However, don't get too excited about breaking free of the same origin restriction; Flash has calling restrictions as well. You can certainly try to put an arbitrary URL in this method, but the Flash Player will first fetch a file from the root of the domain called crossdomain.xml. This file sets up the access policy for remote requests from Flash. For example, http://unsecure.ajaxref.com/crossdomain.xml exists and contains the following rules:

```
<cross-domain-policy>
 <allow-access-from domain="ajaxref.com" to-ports="*"/>
 <allow-access-from domain="*.ajaxref.com" to-ports="*"/>
</cross-domain-policy>
```

This file indicates that other requests from ajaxref.com subdomains can make connections remotely.

The syntax for crossdomain.xml files is quite basic. You have the primary tag `<cross-domain-policy>` that includes `<allow-access-from>` tags. These tags have a domain attribute that is a full domain (for example, www.ajaxref.com), partial wild-card domain (for example, *.ajaxref.com), or full wildcard (*). The secure attribute should be set to true; if set to false, it allows Flash movies served via HTTP to attach to https URLs. The complete DTD for the crossdomain.xml format is shown here.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT cross-domain-policy (allow-access-from*)>
<!ELEMENT allow-access-from EMPTY>
   <!ATTLIST allow-access-from domain CDATA #REQUIRED>
   <!ATTLIST allow-access-from secure (true|false) "true">
```

As we remember, the same origin policy is quite restrictive, and we can't even connect from www.ajaxref.com to unsecure.ajaxref.com with an XHR. With Flash we will be able to do it as long as we have a valid crossdomain.xml on the site we are trying to call, but how does this help us since it requires Flash to be used? It turns out we can bridge calls from JavaScript into a Flash SWF file and back again. Read over the following ActionScript file (ajaxtcrflash.as):

```
import flash.external.ExternalInterface;
class AjaxTCRFlash{
      static function connect(url, callback)
      {
        var myXML = new XML();
         myXML.ignoreWhite = true;
         myXML.onLoad = function(success)
                   {
                   if (success) {
                          ExternalInterface.call(callback, this.toString());
                         }
                   };
```

```
            myXML.load(url);
        }

    static function main()
        {
          ExternalInterface.addCallback("connect", null, connect);
        }
}
```

You should, after your inspection, notice a `connect()` method that takes a `url` and a `callback` that is invoked upon success. This method has been exported to an included Web page as indicated by the line `ExternalInterface.addCallback("connect", null, connect)`.

Now we need to convert this ActionScript into a Flash SWF file. Even if we don't have Flash, there are a number of ActionScript compilers on the Internet to do this. We compiled the example using one called mtasc (www.mtasc.org/):

```
mtasc -version 8 -header 1:1:1 -main -swf ajaxtcrflash.swf ajaxtcrflash.as
```

The 1:1:1 makes the SWF file a 1px × 1px running at 1-frame per second movie. Our goal here is a bit unusual for Flash, to be invisible and behind the scenes to the user.

Next, we take our created SWF file and insert it into the page. The syntax to do this for plug-in-focused browsers like Firefox is primarily using an `<embed>` tag like so:

```
<embed type="application/x-shockwave-flash"    src="http://ajaxref.com/
ch10/flash/ajaxtcrflash.swf" width="1" height="1"  id="helloexternal"
name="helloexternal" />
```

Microsoft's ActiveX component technology would prefer to see the Flash specified like so, using the `<object>` tag:

```
<object id="helloexternal" classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
width="1" height="1" >
  <param name="movie" value="http://ajaxref.com/ch10/flash/ajaxtcrflash.swf" />
</object>
```

Due to some unfortunate lawsuits regarding the use of binary objects within browsers, we have to use script code to insert these elements lest we get a prompt to "Activate this control" in the Microsoft browser. We create a simple function to do just that:

```
function createSWF()
{
  var swfNode = "";
  if (navigator.plugins && navigator.mimeTypes && navigator.mimeTypes.length)
    swfNode = '<embed type="application/x-shockwave-flash" src=
"http://ajaxref.com/ch10/flash/ajaxtcrflash.swf" width="1" height="1"
id="helloexternal" name="helloexternal" />';
  else { // PC IE
         swfNode = '<object id="helloexternal" classid="clsid:D27CDB6E-AE6D-11cf-
96B8-444553540000" width="1" height="1" >';
         swfNode += '<param name="movie" value="http://ajaxref.com/ch10/flash/
ajaxtcrflash.swf" />';
```

```
        swfNode += "</object>";
      }
   /* put the Flash reference in the page */
  document.getElementById("flashHolder").innerHTML = swfNode;
}
```

NOTE *Insertion and manipulation of Flash movies is filled with all sorts of little details. Many developers rely on scripts like SWFObject (http://blog.deconcept.com/swfobject/) to perform such tasks. Our point here is demonstration, and the approach taken should work for most readers.*

Once our communications SWF file is inserted into the page, we find the Flash movie and then use its externally exposed `connect()` method to make our call to a URL and specify the callback we want to use. Of course, nothing can be the same between the browsers. We see accessing the SWF object is a bit different, so we write a little function to abstract that as well:

```
function getSWF(movieName)
 {
   if (navigator.appName.indexOf("Microsoft")!= -1)
      return window[movieName];
   else
     return document[movieName];
  }
flashBridge = getSWF("helloexternal");
```

Finally, after getting a handle to the Flash object we issue the request:

```
flashBridge.connect("http://unsecure.ajaxref.com/ch1/sayhello.php ",
"printMessage");
```

This will later call `printMessage` and show us content from another domain! Figure 10-4 shows the demo at http://ajaxref.com/ch10/flashajax.html breaking the same origin policy. The complete code that enabled this is shown next for your perusal.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Chapter 10 - Breaking SOP with Flash</title>
<script type="text/javascript">
      function createSWF()
      {
            var swfNode = "";
            if (navigator.plugins && navigator.mimeTypes && navigator
.mimeTypes.length)
                  swfNode = '<embed type="application/x-shockwave-flash"
src="http://ajaxref.com/ch10/ajaxtcrflash.swf" width="1" height="1"
id="helloexternal" name="helloexternal" />';
            else {
                  swfNode = '<object id="helloexternal" classid=
"clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" width="1" height="1" >';
                  swfNode += '<param name="movie" value=
```

```
"http://ajaxref.com/ch10/ajaxtcrflash.swf" />';
                  swfNode += "</object>";
            }
            document.getElementById("flashHolder").innerHTML = swfNode;
      }

      function getSWF(movieName)
      {
        if (navigator.appName.indexOf("Microsoft")!= -1)
         return window[movieName];
        else
         return document[movieName];
       }
      function printMessage(str)
      {
        document.getElementById("responseOutput").innerHTML = str;
       }
      window.onload = function()
      {
        createSWF();
        document.getElementById("helloButton").onclick = function(){
         var flashBridge = getSWF("helloexternal");
         flashBridge.connect("http://unsecure.ajaxref.com/ch1/
sayhello.php", "printMessage");  }
      }
</script>
</head>
<body>
<form action="#">
 <input type="button" value="Say Hello" id="helloButton" />
</form>
<br /><br />
<div id="flashHolder"></div>
<div id="responseOutput"> </div>
</body>
</html>
```

You may want to note a couple of items in Figure 10-4. First, you can clearly see the fetch for the crossdomain.xml file before the request is invoked. Second, the continuous status message presented to the user when Flash remoting is used, which might be a bit disconcerting to users.

The Future: Native XHR Cross Domain Access

In the very near future, maybe even as you read this, it is quite likely that browsers will more commonly break the same origin policy (SOP) and boldly go where no XHR has gone before. Early versions of Firefox 3 include the first attempt at native XHR cross-domain access and have implemented the emerging W3C standard for cross-site access control (www.w3.org/TR/access-control/). Following this specification to enable cross-site access, the resource in question has to issue an access control header in its response.

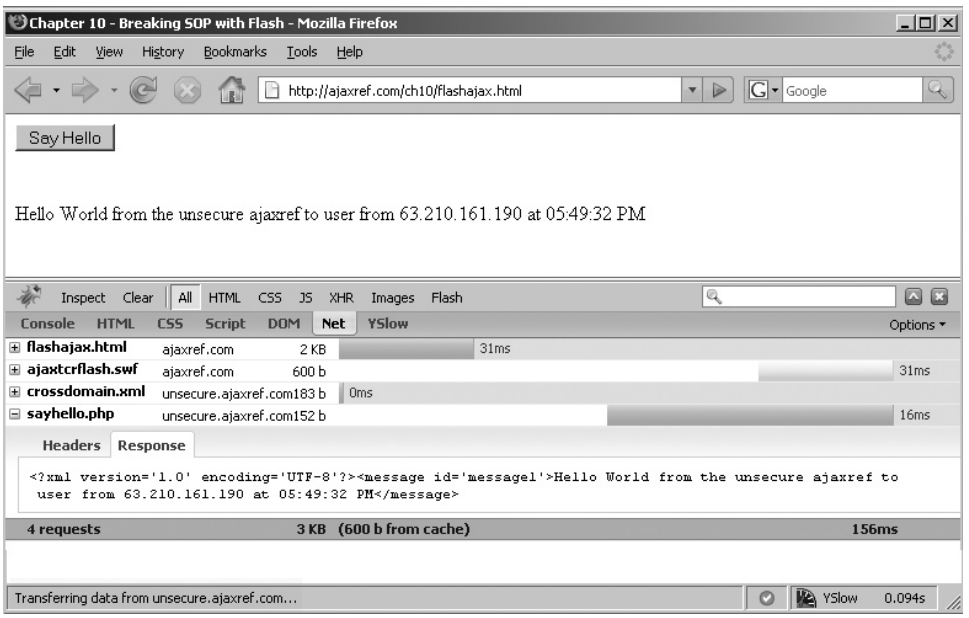


FIGURE 10-4 Flash going where many XHR implementations fear to tread!

This is somewhat similar to crossdomain.xml but a bit more granular since it can be used on a file-by-file basis. For example, we might issue a header in our response like:

```
Content-Access-Control: allow <*>
```

This says the resource can be attached by anyone from any domain. To be a bit less permissive, we might limit it to requests from a particular set of domains with a response like so:

```
Content-Access-Control: allow <ajaxref.com>
```

or even limit it to requests from a particular set of domains with exclusions:

```
Content-Access-Control: allow <ajaxref.com> <*.ajaxref.com> exclude
<unsecure.ajaxref.com>
```

If the content items are generated, it is fairly easy to set these kinds of rules, but if we are serving static files it might be a bit difficult to get them in place. You would likely have to put the remotely accessible files in a particular directory and then set rules on your Web server, for example using Apache's mod_headers. However, the current specification does provide one instance where that is not the case, serving XML files. In this case a processing directive can also be used to specify the same kind of rule.

```
<?xml version='1.0' encoding='UTF-8'?>
<?access-control allow="*"?>
<packet>
<message id="message1">To boldly go where no XHR has gone before...</message>
</packet>
```

From a coding point of view there really isn't anything to go client side. We should be able to issue a request as we normally would.

```
var url =  "http://some-other-site-that-allows-remote-access/servicecall";
var options = {method:"GET",
               onSuccess : handleResponse};
AjaxTCR.comm.sendRequest(url, options);
```

Unfortunately, as we test this, we note that the way it is handled is currently incompatible with not only our library, but also with other libraries like YUI and Prototype. It is quite likely that wrapping the XHR invalidates the request as they may be considering XHR hijacking. However, it is also quite likely that this is simply very alpha technology. However, going back to our Chapter 3 knowledge we can do things manually like so:

```
var xhr = new XMLHttpRequest();
xhr.open('GET','http://unsecure.ajaxref.com/ch10/sayhello.php',true)
xhr.onreadystatechange = function (){handleResponse(xhr)};
xhr.send(null);
```

This will work just fine, as shown in Figure 10-5. When you are armed with the Firefox 3 browser, check the example at http://ajaxref.com/ch10/crossdomainxhr.html to see if you too can break the SOP!

SOAP: All Washed Up?

If you are a Web Services aficionado, you might get a whiff of RESTful bias here, given all the examples presented up until now. Certainly SOAP (Simple Object Access Protocol) has been practically synonymous with Web Services in the past, but that does not seem to be the case for public-facing Web Service APIs. In fact, fewer and fewer of them seem to be supporting SOAP (see the upcoming section "Sampling Public Services"), probably due to complexity and the lack of native browser implementations. Interestingly on that front, the most notable SOAP-aware browser, Firefox, appears to be planning to remove SOAP from its 3.0 release. Does this mean that SOAP is all washed up, at least in terms of end-user Web Service use? Actually no, if we consider that SOAP is just an XML format. Why couldn't we use JavaScript to make the packet and then use standard Ajax to make the call?

SOAP can easily live on within an XHR-powered world. For example, notice in the following example how we manually make a SOAP packet, stamp the correct content type on it, and send it on its way to a SOAP service.

```
function sendRequest()
{
  var url = "http://ajaxref.com/ch10/soapserver.php";
  var payload = '<?xml version="1.0" encoding="UTF-8"?>' +
                              '<SOAP-ENV:Envelope  xmlns:SOAP-ENV="http://schemas.
xmlsoap.org/soap/envelope/"' +
                              'xmlns:xsd="http://www.w3.org/2001/XMLSchema"' +
                              'xmlns:xsi="http://www.w3.org/2001/XMLSchema-in-
stance"' +
                              'xmlns:SOAP-ENC=
"http://schemas.xmlsoap.org/soap/encoding/"' +
```

FIGURE 10-5 SOP busted natively!

```
                              'xmlns:ns4="urn:helloworld"' +
                              'SOAP-ENV:encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/">' +
                              '<SOAP-ENV:Body>' +
                              '<ns4:helloworld>' +
                              '</ns4:helloworld>' +
                              '</SOAP-ENV:Body>' +
                              '</SOAP-ENV:Envelope>';
  /* define communication options */
  var options = { method: "POST",
                  onSuccess : handleResponse,
                 requestContentType: "text/xml",
                  payload: payload
                };
  AjaxTCR.comm.sendRequest(url,options);
}
```

The service handles our "helloworld" call and responds with our favorite welcoming message via a SOAP response.

```
<?php
  function helloworld()
  {
    return "Hello World to user from " . $_SERVER['REMOTE_ADDR'].
" at ". date("h:i:s A");
  }
  $server = new SoapServer(null, array('uri' => "urn:helloworld"));
  $server->addFunction("helloworld");
  $server->handle();
?>
```

Back on the browser we then receive the packet and parse it putting the payload in the page.

```
function handleResponse(response)
{
  var result = response.responseXML.getElementsByTagName("return")
  $id("responseOutput").innerHTML = result[0].firstChild.nodeValue;
}
```

The operation and network trace of this SOAP example is shown in Figure 10-6, and the example can be found at http://ajaxref.com/ch10/soapclient.html.

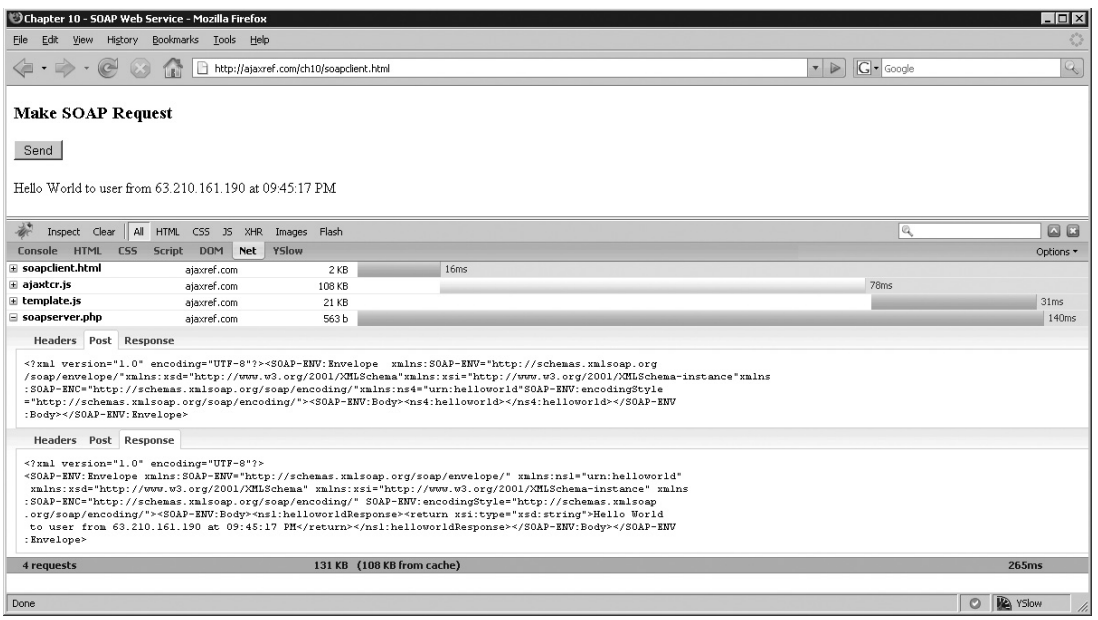
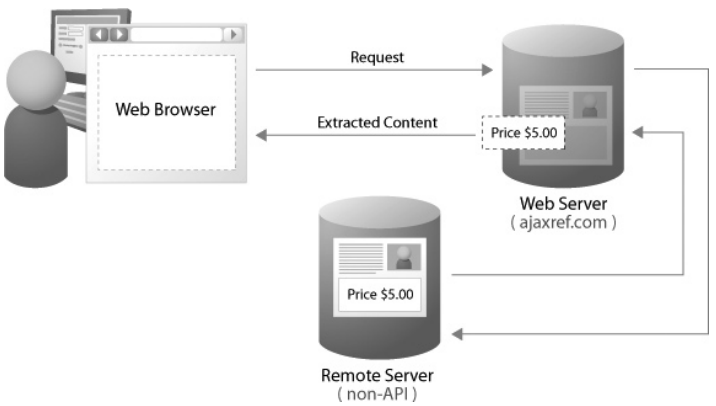


FIGURE 10-6 SOAPy Ajax request

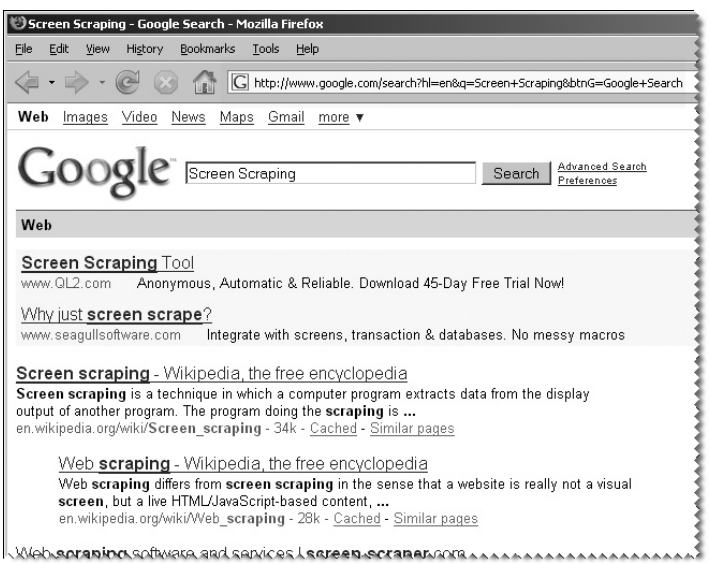
No doubt the communications process could be abstracted so that you could form SOAP packets more programmatically in JavaScript, but our point here is simply that Web Services using SOAP can certainly live in the world of Ajax.

Screen Scraping

Sometimes public sites don't provide clear APIs for programmers. In these cases, developers interested in using the data or services provided by the site resort to an idea called screen scraping. The basic sense of screen scraping is to browse the site literally as a normal human browser would, fetch the HTML and other resources, and then extract the pieces of interest to use in their own way—for good or for ill.



To use a simple example, let's issue a query at Google:



and then inspect the query string:

```
http://www.google.com/search?hl=en&q=Screen+Scraping&btnG=Google+Search
```

It is clear from this that we change the query easily enough to the more technically appropriate term "Web Scraping," like so:

```
http://www.google.com/search?hl=en&q=Web+Scraping&btnG=Search
```

Since that is all we need to do to alter a search, it would seem we could automate the trigger of a Google search quite easily. For example, in PHP we might simply do:

```
$query = "screen+scraping";  // change to whatever
$url =  "http://www.google.com/search?hl=en&q=$query&btnG=Google+Search";

$result = file_get_contents($url);
```

Now in `$result` we are going to get a whole mess of HTML, like so:

```
<html><head><meta http-equiv=content-type content="text/html; charset=UTF-8">
<title>Screen Scraping - Google Search</title><style>div,td,.n a,.n a:
visited{color:#000}.ts
... snip ...

<div class=g><!--m--><link rel="prefetch" href="http://en.wikipedia.org/
wiki/Screen_scraping"><h2 class=r><a href="http://en.wikipedia.org/wiki/
Screen_scraping" class=l onmousedown="return clk(0,'','','res','1','')"><b>
Screen scraping</b> - Wikipedia, the free encyclopedia</a></h2><table bor-
der=0 cellpadding=0 cellspacing=0><tr><td class="j"><font size=-1><b>Screen
scraping</b> is a technique in which a computer program extracts data from
the display output of another program. The program doing the <b>scraping
</b> is <b>...</b><br><span class=a>en.wikipedia.org/wiki/<b>Screen</b>_
<b>scraping</b> - 34k - </span><nobr>

...snip...
```

We could try to write some regular expressions or something else to rip out the pieces we are interested in, or we might rely on the DOM and various XML capabilities available. Most server-side environments afford us better than brute force methods, so we instead load the URL and build a DOM tree.

```
$dom = new domdocument;
/* fetch and parse the result */
$url = 'http://www.google.com/search?hl=en&q=screen+scraping&btnG=Google+Search';
@$dom->loadHTMLFile($url);
```

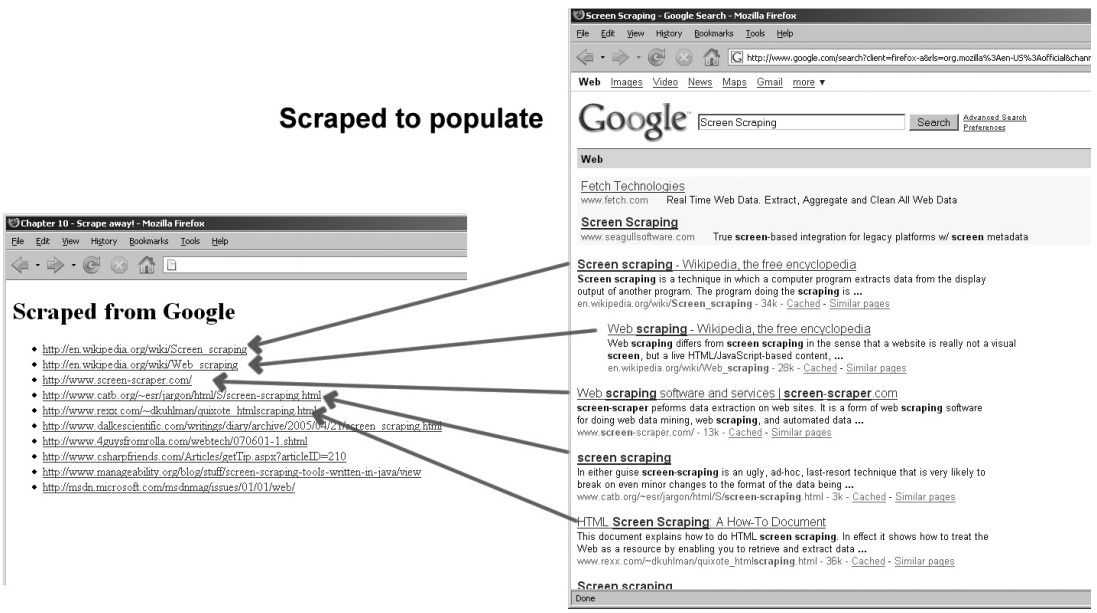
Then we take the DOM tree and run an Xpath query on the results to rip out what we are interested in, in this case some links. After having inspected the result page, it appears that the good organic results have a class of "l" (at least at this point in time), so we pull out only those from the result.

```
/* use xpath to slice out some tags */
$xpath = new domxpath($dom);
$nodes = $xpath->query('//a[@class="l"]');
```

Finally, we print out the resulting nodes to our own special results page without ads and other items:

```
/* print out the tags found */
print "<ul>";
foreach ($nodes as $node)
{
    $resultURL = $node->getAttribute('href');
    if ($resultURL != '')
     echo "<li><a href='$resultURL'>$resultURL</a></li>";
}
print "</ul>";

?>
```

We can see the working result here:



Note we don't give a URL for you to try because, frankly, the demo is likely to fail sometime in the very near future, especially if Google changes its markup structure or they ban us for querying too much.

Scraping is fragile, and scraping grab content or mash-up data that is not to be used without surrounding context is certainly bad practice. However, the technology itself is fundamental to the Web. We need to be able to automate Web access, for how else would Web testing tools work? We present the idea only to let you know that scraping might be a necessary evil to accomplish your goals in some situations.

If after reading this you are concerned about scraping against your own site, the primary defense for form-based input would be a CAPTCHA (http://en.wikipedia.org/wiki/Captcha) system, as shown here, where the user types the word shown into some text box for access:

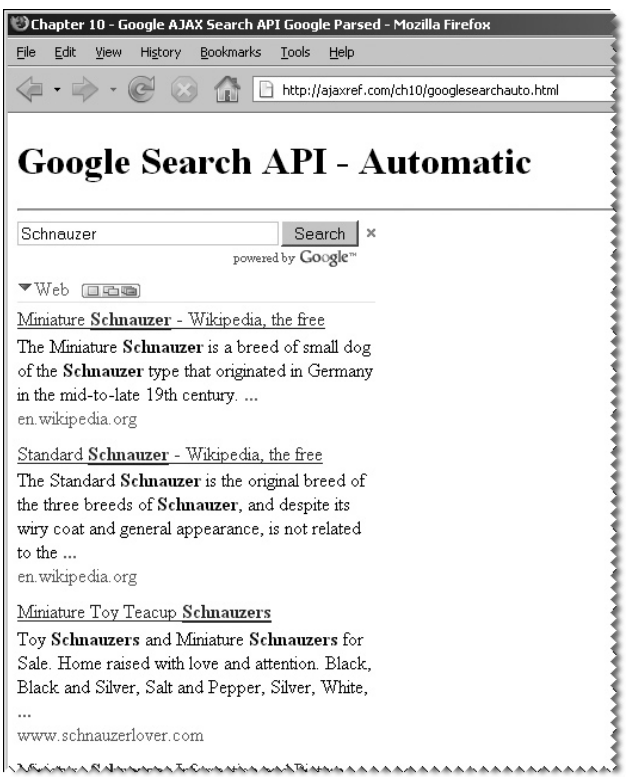
Of course, as this example shows you need to balance what is difficult for a bot to solve with what a human can actually read.

When trying to protect content it would make sense to try other schemes such as randomization of markup structure including `id` and `class` values. You might even decide to put content in less scrapable formats for example putting textual content in a binary. Ultimately though, content wise if the user can view it, they can get it and can likely automate it. To keep out automated content scraping, you would just have to monitor for the frequent access from set IPs and then ban any bot command hooligans who start abusing your site or application.

Sampling Public Services

In this section we take a brief moment to review public Web Services available when this book was written. The goal here is not to present a cookbook of usage. Very likely you will need to visit the services in question for the latest information on syntax and access policies. Rather, our goal in showing a few examples is to illustrate the range of possibilities, as well as the typical logistic and technical requirements that will be faced when dealing with public Web Services.

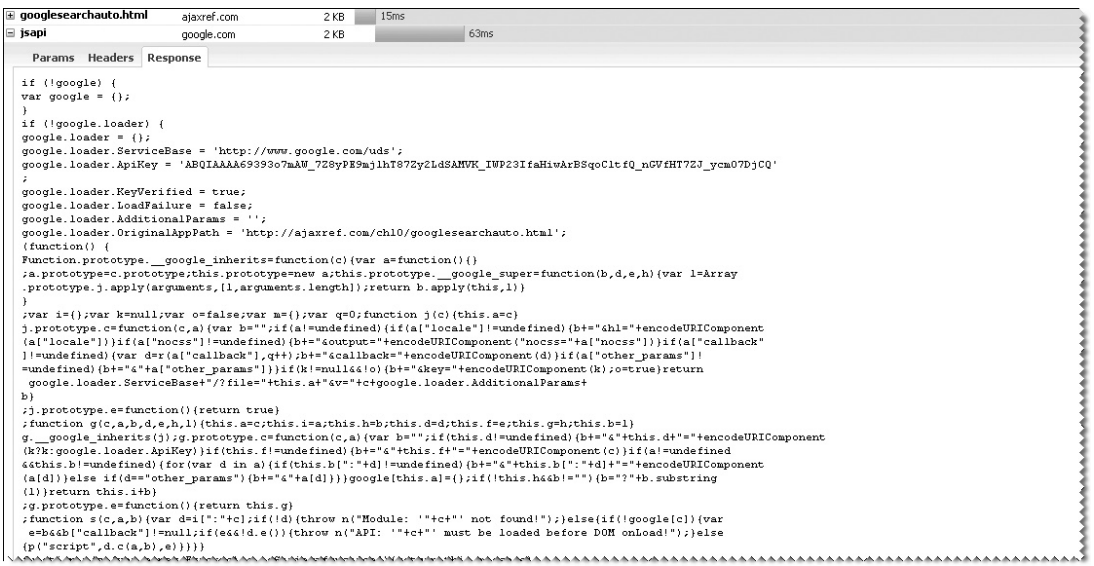
The first services explored are the Google APIs for search feeds and maps. Information about each service can be found at http://code.google.com. The first example shows a simple version of the Google Search API to load in a simple query box that will retrieve search results in page, Ajax style.



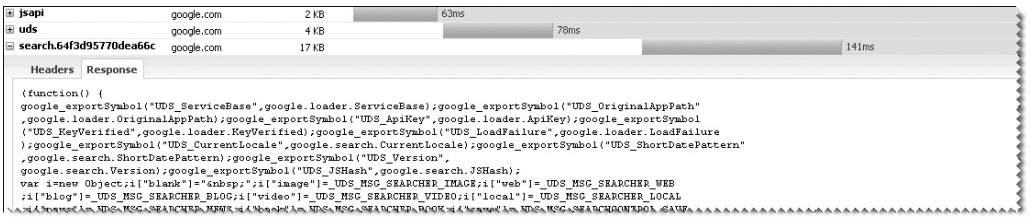
Now let's see how this works and why we said "Ajax style." First, we note the inclusion of a `<script>` tag in our example page with an API key.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Chapter 10 - Google AJAX Search API Google Parsed </title>
<!--
 Do not use this Google API key as it only works on this site
 and in this directory.
-->
<script src="http://www.google.com/jsapi?key=ABQIAAAA69393o7mAW_
7Z8yPE9mj1hT87Zy2LdSAMVK_IWP23IfaHiwArBSqoC1tfQ_nGVfHT7ZJ_ycmO7DjCQ" type=
"text/javascript"></script>
```

Like most public Web Services, to avoid abuse Google makes you register for an access key to use their services. What is interesting about this call is that it is also a bootstrap mechanism that is loading in the script that powers this facility.



That isn't the only file we see; others are pulled in as well during the process:

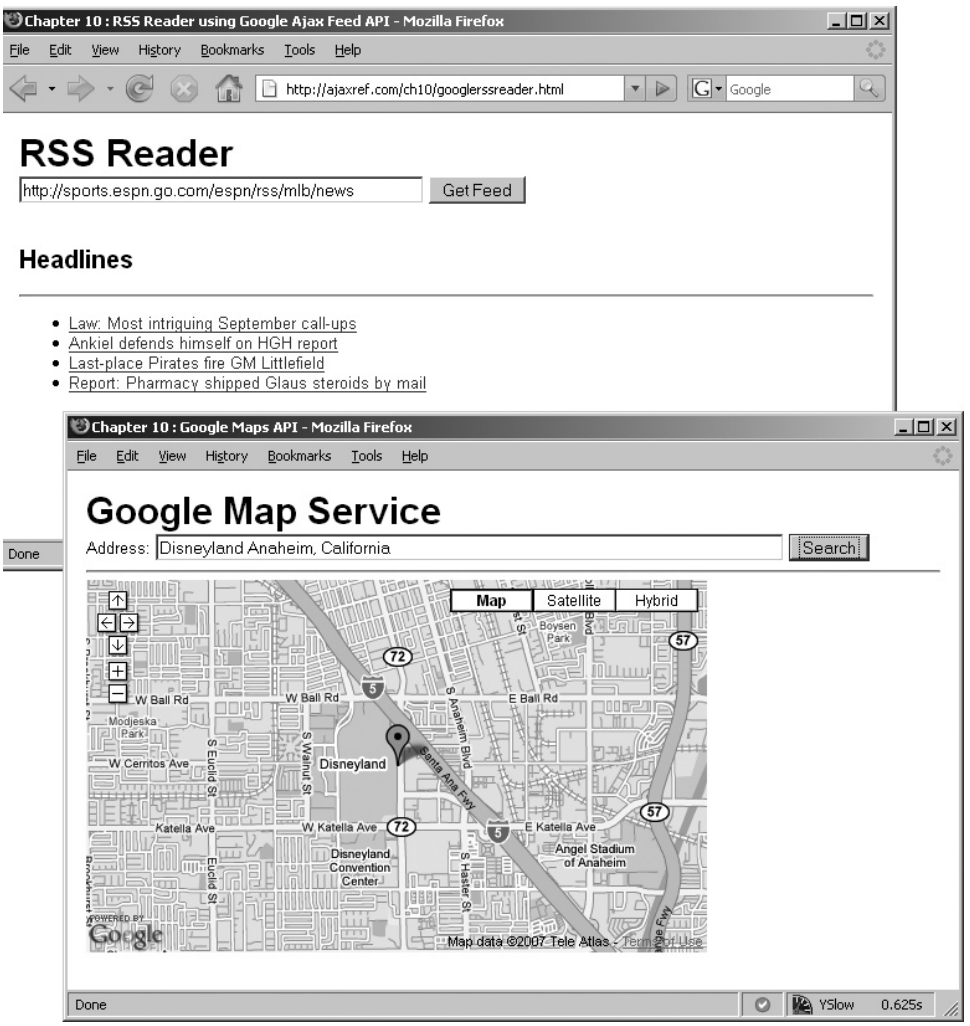


What's interesting here is Google's approach of creating a generic loader so they can pull in new versions of code quite easily, rather than having the user point to some new filename.

Now client side, you must add some code to enable the Google search but it is pretty minimal. We load the Google search service, instantiate and add a `google.search.SearchConrol` object to the page, define some parameters, and make sure to bind it to a `<div>` element in our layout.

```
<script type="text/javascript">
google.load("search", "1");
window.onload = function () {
  var searchControl = new google.search.SearchControl();
  var options = new google.search.SearcherOptions();
  options.setExpandMode(google.search.SearchControl.EXPAND_MODE_OPEN);
  searchControl.addSearcher(new google.search.WebSearch(), options);
  searchControl.setResultSetSize(google.search.Search.LARGE_RESULTSET);
  searchControl.draw(document.getElementById("searchcontrol"));
};
</script>
</head>
<body>
<h1>Google Search API - Automatic</h1>
<hr />
  <div id="searchcontrol">Loading...</div>
</body>
</html>
```

And now we have an in-page Google powered search box (http://ajaxref.com/ch10/googlesearchauto.html). Yet this isn't Ajax powered in the strict sense of an XHR. In fact, if you try the other services Google offers like Maps (http://ajaxref.com/ch10/googlemap.html) and the RSS feed reader (http://ajaxref.com/ch10/googlerssreader.html), you'll see the same thing:

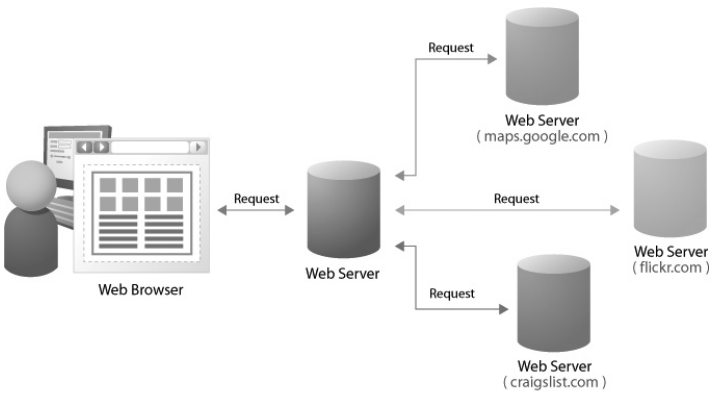


The situation will be no different for other public Web Services found. If you are looking for direct consumption in a client, it will almost certainly be JSON or script responses invoked by `<script>` tag insertions and not using any sort of XHR mechanism given their same origin restrictions. Besides Google, you will find all sorts of services from sites like Yahoo, eBay, Amazon, and many others. A very complete list of Web APIs can be found at www.programmableweb.com/apis.

Mash-ups

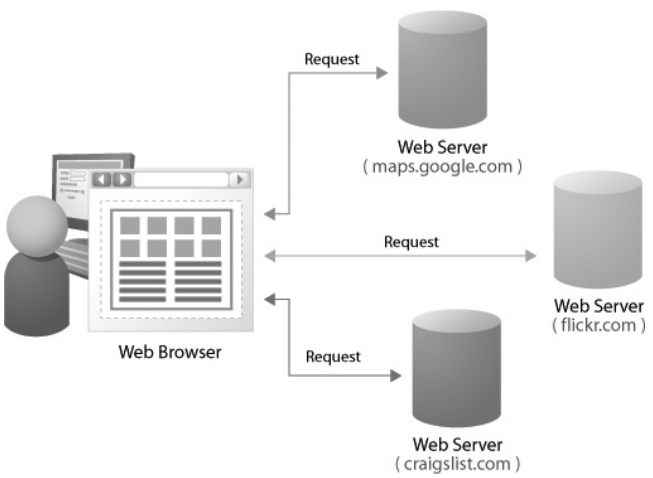
With all these various Web Services providing interesting data online, it would seem we could build valuable aggregates by combining and correlating data fetched from various services into a new page. This concept is what is popularly termed a mash-up. Now, as we

have seen with our exploration of Web Services and Ajax, we will very likely use a proxy to fetch data, so the actual mashing will likely occur on the proxying server.



Using a proxy

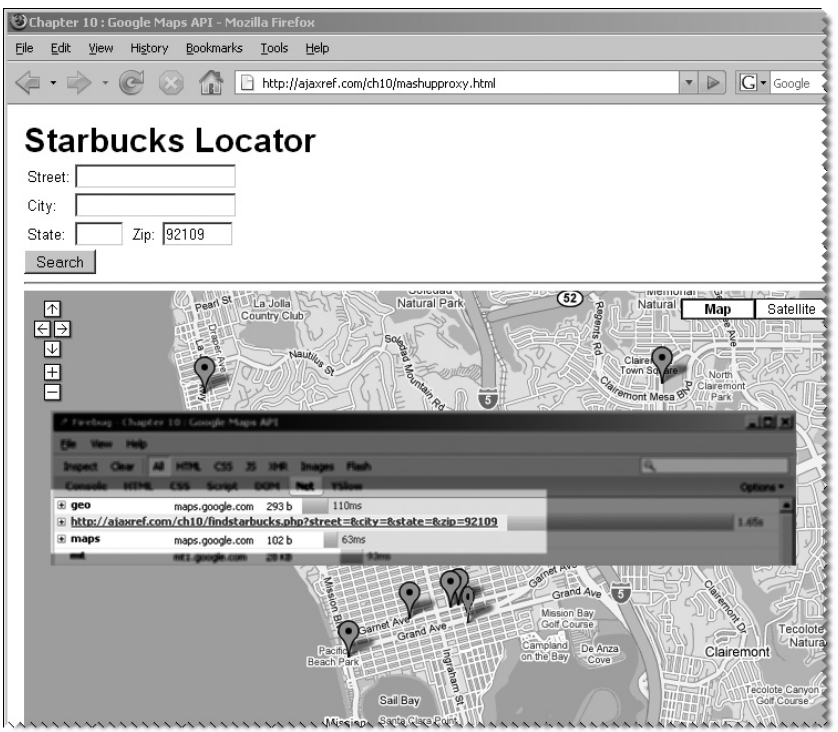
Of course, given the possibility of using `<script>` tags with JSON responses for direct access, it might be possible to do an in-browser mash-up as well.



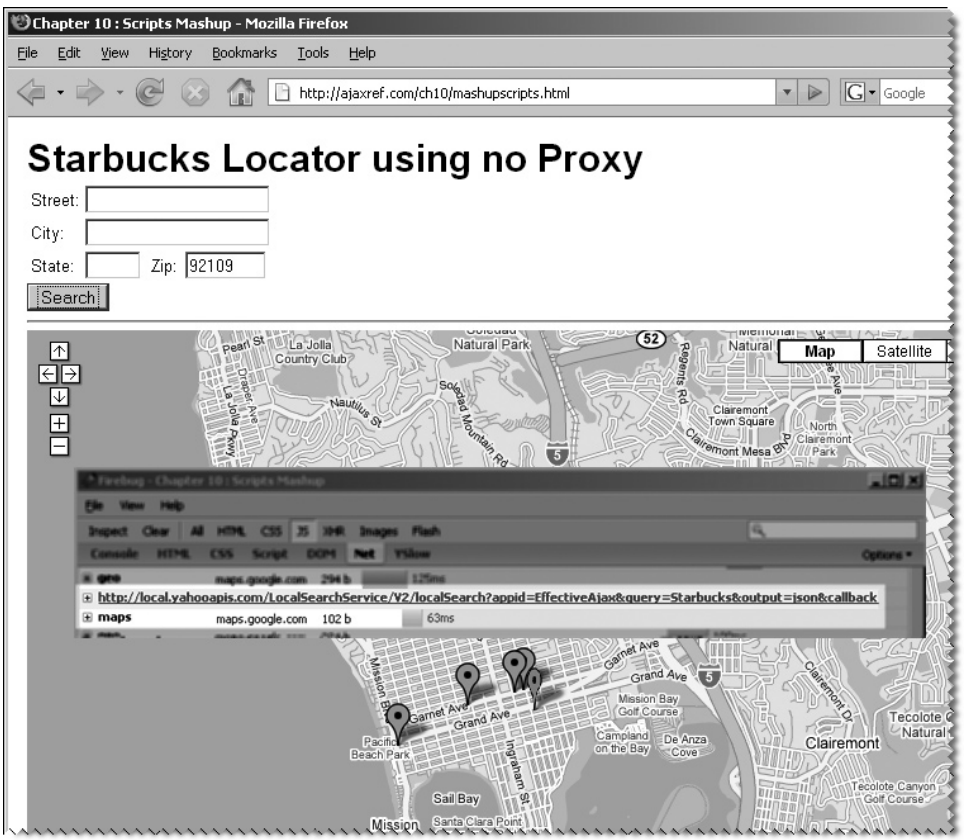
Direct using script tag or other mechanism

It is also possible to use combinations of direct `<script>` calls and proxy calls.

As an example, we built a simple mash-up that allows you to type in an address. It fetches a map from Google Maps and combines the data with the local Starbucks in your vicinity in case you are in dire need of corporate caffeine. In the version at http://ajaxref.com/ch10/mashupproxy.html, it pulls the data from Google Maps directly via a `<script>` tag approach but uses a PHP proxy to fetch the store location via a Web scrape and then combine them together.



In the second version at http://ajaxref.com/ch10/mashupscripts.html, we pull our data using a `<script>` call to Google and Yahoo and then combine the two.



The code for either example is more busy work than complex. Mash-up code mostly involves fetching data in a variety of ways, translating data from one format to another, and then combining the interesting items. Given the consistency of approach, a number of efforts have been made to build visual mash-up creation tools. For example, http://pipes.yahoo.com, as shown in Figure 10-7, is used to create a simple mash-up that reads a number of popular Ajax news source sites and then provides a query mechanism against the stories.

While making mash-ups can be fun, we encourage you to look at mash-up making systems or simply look at the list of existing efforts, as it is very likely the combination of data or something quite similar has been done before.

Comet

For a more continuous connection to the server in order to keep the client up to date, an Ajax application must rely on a polling mechanism to make requests to check status at regular intervals. This approach can be quite taxing on server and client alike. For irregularly

FIGURE 10-7 Plumbing Web 2.0 with pipes

occurring events, this approach is quite inefficient and is completely unworkable for approaches that need a real-time or near real-time connection. The Comet communications pattern changes this by keeping a connection open between the browser and server so that the server can stream or push messages to the browser at will, as shown in Figure 10-8.

NOTE *Comet is not an acronym and appears to be a somewhat tongue-in-cheek cleaner-related moniker given to a collection of server-push approaches being used. The introduction of the term is attributed to Alex Russell of Dojo Toolkit fame around March 2006. The implication of how this pattern was implemented, coupled with dislike of the expression, has led others to introduce a variety of other terms of similar meaning for commercial or personal reasons, much to the confusion of developers and Ajax book authors alike.*

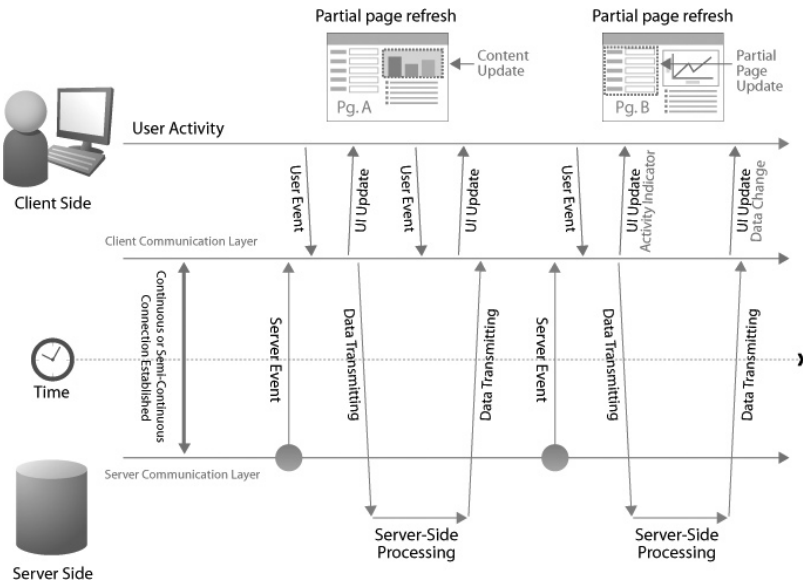


Figure 10-8 Comet, push reborn

What to call this push-oriented communication pattern and how exactly it should be accomplished is subject to much debate and confusion. A continuous polling mechanism certainly doesn't count, but if the frequency were enough that it would provide the effective functionality for most applications—we'll dub that the *fast poll*. Another approach would be to use a *long poll*, where an XHR is employed and holds a connection open for a long period of time and then re-establishes the poll every time data is sent or some timeout is reached. Still another approach is often dubbed the *slow load* or the "endless iframe," given how it is usually implemented as a continuous connection sustained through a connection that never terminates. We might also introduce true two-way communication using a socket connection bridged from a Flash file or Java applet into the page—we call that a *binary bridge*. Finally, given the need for real-time event handling, some browsers have introduced *native server-event monitoring*. All the approaches are summarized in Table 10-2 and shown visually in Figure 10-9.

We present each of the communication schemes individually to explore their implementation and network traces before taking a brief look at everyone's favorite sample push-style application: chat.

Approach	Description	Comments
Fast poll	Calls the server very rapidly using a standard XHR call to see if changes are available.	Uses standard HTTP request to a Web server. Not really a push but if continuous enough appears as instantaneous. Significant burden on server and network with numerous requests. No way for server to initiate the data transfer.
Long poll	Uses an XHR, but we hold the connection open for an extended period of time say 20-30 seconds. After the time threshold is reached, the connection is shut down and re-established by the client. The server may push data down the held connection at any time and thus shut the connection, which the browser will immediately re-open.	Uses standard Web server with HTTP connections. Server can push data to browser assuming there is a held connection open. Held connections and some Web server-application server architectures may not get along well. Gap of no connectivity when browser re-establishes connection after data transfer or timeout.
Slow load	Uses an iframe that points to a never finishing URL. The URL in question is a program that pushes data when needed to the iframe, which then can call upward into the hosting page to provide the newly available data.	Does not use an XHR and thus lacks some networking and script control, though as an iframe it works in older browsers. Continuous load can present some disturbing user interface quirks such as a never finishing loading bar. Tends to result in growing browser memory consumption and even crashes if connection held upon for a very long time.
Binary bridge	Uses Flash or Java applet to make a socket connection to the server. As two-way communication, the socket provides full push possibilities. Received data is made available via a JavaScript from the communications helper binary.	Relies on binary that may not be installed. Piping between JavaScript and binary may be problematic. Very flexible in terms of communication methods and data formats.
Native browser access	In some browsers like Opera 9 you should be able to subscribe to server events that will wake the browser when data is made available.	Uses native browser facilities. Apparently works similarly to an endless iframe from a network point of view. Not widely implemented as of yet.

Table 10-2 Summary of Push-style Communications Approaches

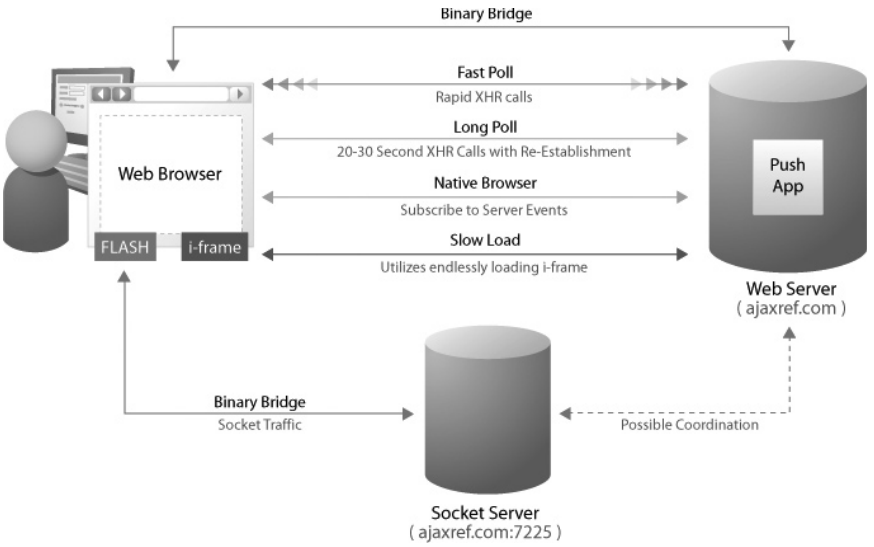
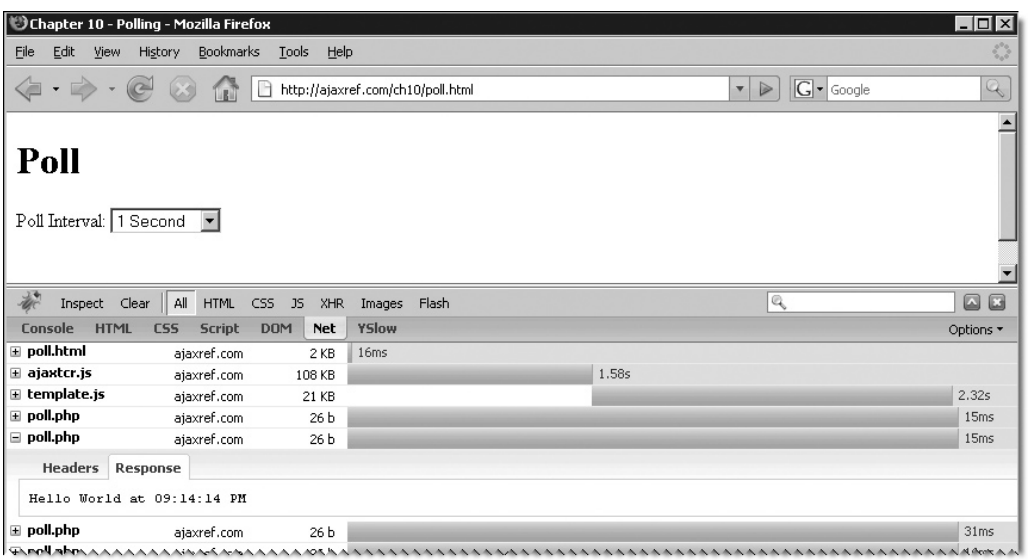


FIGURE 10-9 Many different approaches to Comet or push-style communication

Polling: Fast or Long

The polling pattern may not be graceful, but it is effective in a brute force manner. Using a timer or interval we can simply repoll the server for data.



If the polling frequency is fast enough, it can give a sense of immediate data availability (see http://ajaxref.com/ch10/poll.html). However, if little activity occurs, you end up issuing

a great number of network requests for very little value. You might consider adding a decay concept to a polling solution, the idea being that if you do not see changes you increase the delay between poll attempts. However, a downside to this approach is that when such infrequent changes do happen, it may be some time before the user is altered to them.

The long poll pattern is better for dealing with updates that may not be predictable. Connections are re-established upon data or can be set to re-establish upon a timeout with a retry mechanism. The following example (http://ajaxref.com/ch10/longpoll.html) uses this pattern to call a server-side program that responds with a varying amount of time. If the server doesn't respond in 30 seconds, it will retry again for a total of 10 times, assuming a three-minute period of inactivity indicating the server being unavailable. However, if the server does respond, you'll note that `outputTarget` gets updated, but the `onSuccess` handler just starts the request all over again.

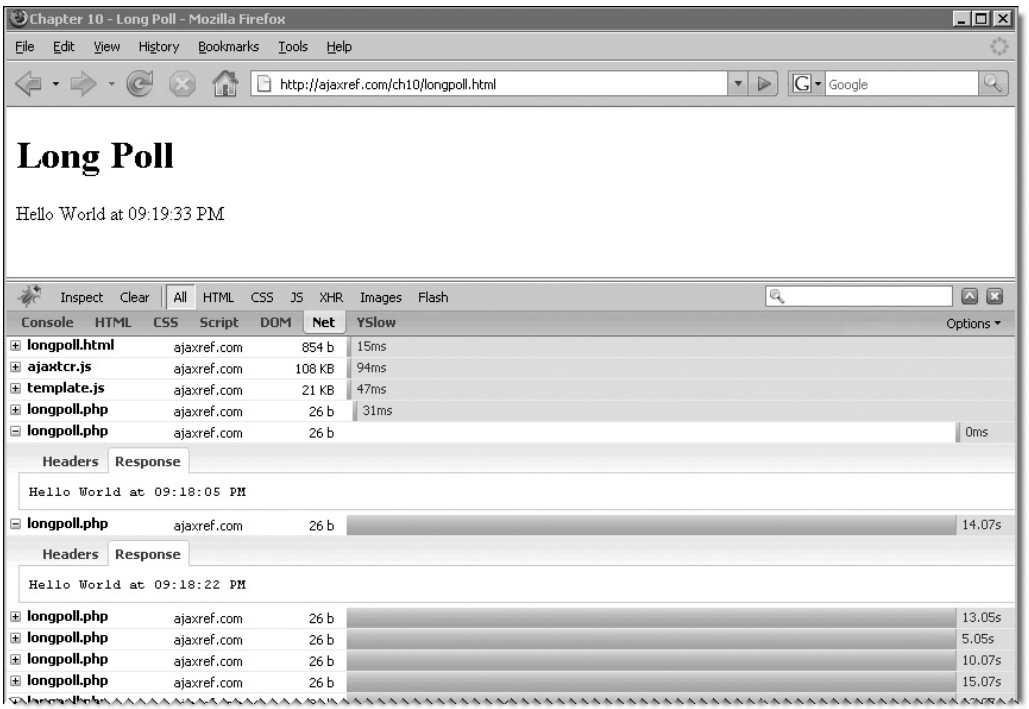
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Chapter 10 - Long Poll</title>
<script src="http://ajaxref.com/ch10/ajaxtcr.js" type="text/javascript"></script>
<script type="text/javascript">
function sendRequest(response)
{
   var options = {method: "GET",
                  outputTarget: "hellodiv",
                  retries: 10,
                  timeout: 30000,
                  onSuccess: sendRequest};

   /* treat the first response specially - no delay */
   if (!response)
     options.payload = "delay=0;";
   AjaxTCR.comm.sendRequest("http://ajaxref.com/ch10/longpoll.php", options);
}
AjaxTCR.util.event.addWindowLoadEvent(function(){sendRequest(false);});
</script>
</head>
<body>
<h1>Long Poll</h1>
<div id="hellodiv"></div>
</body>
</html>
```

The simple PHP code to simulate a long poll pattern just creates random delays to give a sense of intermittent server activity.

```
<?php
header("Cache-Control: no-cache");
header("Pragma: no-cache");
if ($_GET["delay"])
  $delay =$_GET["delay"];
else
```

```
  $delay = rand(1,20);
sleep($delay);
print 'Hello World at ' . date("h:i:s A");
?>
```

The network trace here shows the long poll pattern in action:



NOTE *Close- and timer-based re-establishment of connections is not limited to an XHR communication; iframes or other transports can use a similar mechanism.*

The Long Slow Load

For many, the long slow load pattern or endless iframe is what they think of when the term Comet is used. We demonstrate here making an iframe connection to a server-side program, indicating where we want the response data to be placed in this case a `<div>` named "hellodiv."

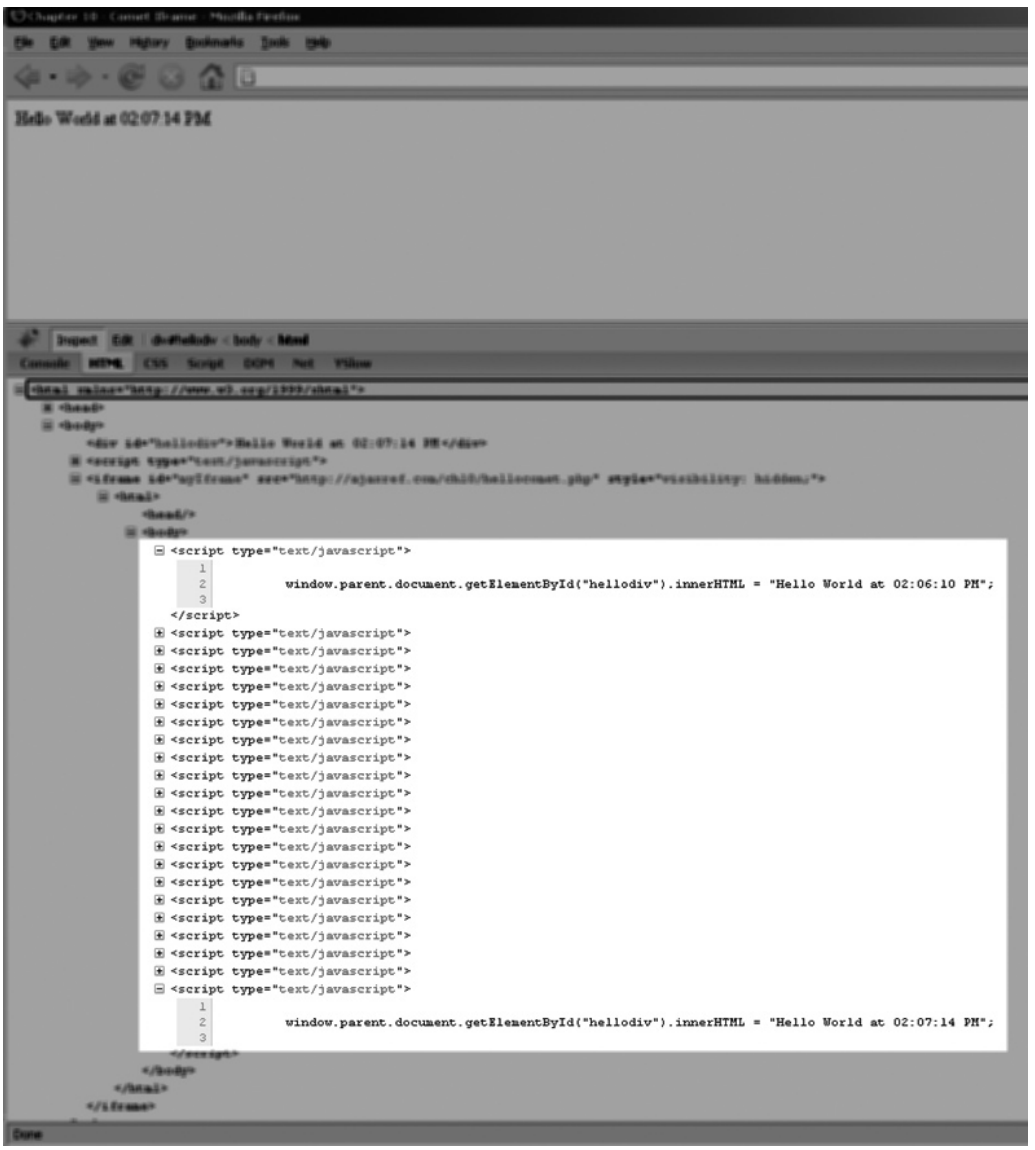
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

```
<title>Chapter 10 - Comet Iframe</title>
<script src="http://ajaxref.com/ch10/ajaxtcr.js" type="text/javascript"></script>
<script type="text/javascript">
function sendRequest()
{
 var options = {method: "GET",
                transport: "iframe",
                payload : "output=hellodiv"};
 AjaxTCR.comm.sendRequest("http://ajaxref.com/ch10/endlessiframe.php", options);
}
AjaxTCR.util.event.addWindowLoadEvent(sendRequest);
</script>
</head>
<body>
<div id="hellodiv"></div>
</body>
</html>
```

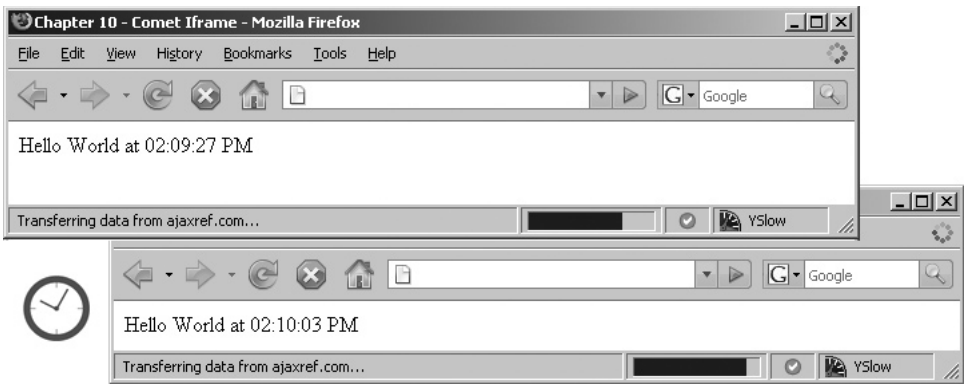
On the server we generate a response page to go in the iframe transport. We first notice the code outputs a `<script>` tag that will call the parent window and put content in the specified DOM element found in `$output`, which in our case is "hellodiv." We also note that it does this output in an endless loop and flushes the contents out in two-second intervals.


```
<?php
header("Cache-Control: no-cache");
header("Pragma: no-cache");
?>
<html>
<head>
<title>No Title Required!</title>
</head>
<body>
<?php
  $output = $_GET["output"];
  while ($output)
    {
     print '<script type="text/javascript">';
     print 'window.parent.document.getElementById("' . $output . '").innerHTML =
"Hello World at ' . date("h:i:s A") . '";';
     print '</script>';
     ob_flush();
     flush();

     sleep(2);
    }
?>
</body>
</html>
```

Back in the browser, the time is updated every few moments, but looking at the DOM we see a whole bunch of `<script>` tags being added into the never-ending iframe:

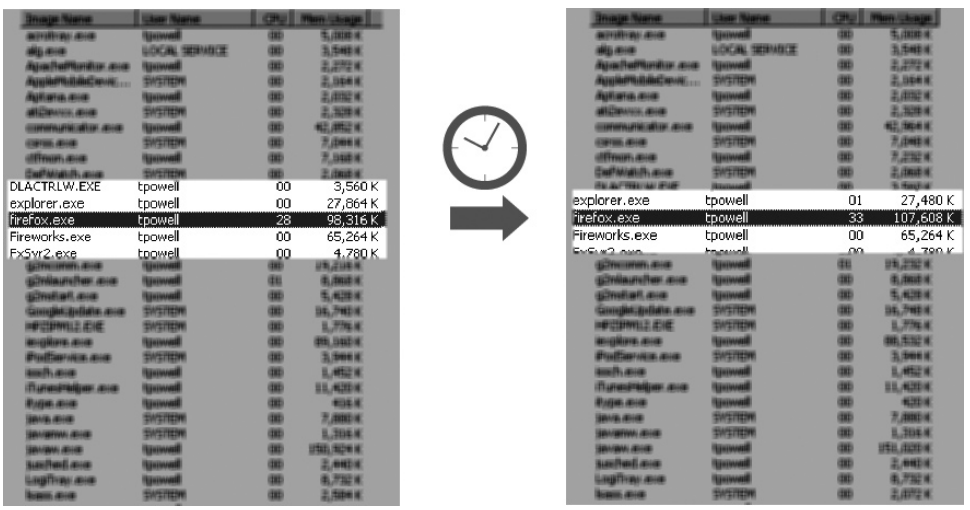


We also note that the browser loading part makes it look like we are never finished loading the page:



Some have argued this UI quirk is a good thing because it lets the user know they have a connection, but we think that is overly optimistic view of how users will interpret that indicator.

Finally, we note that if we let the example run for a while the browser's memory foot print will grow and grow.

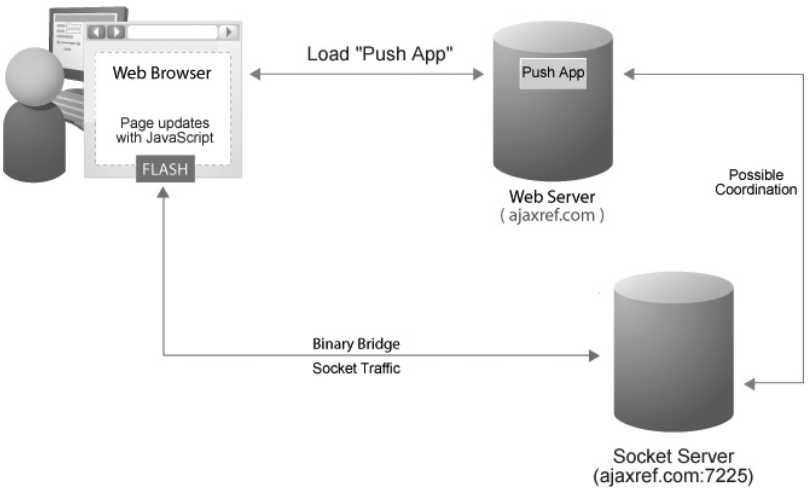


The long slow load may have its issues, but it does work. Give it a try yourself at http://ajaxref.com/ch10/endlessiframe.html.

Binary Socket Bridget

When Ajax needs a little help from its friends, embedded binaries like Flash or Java can be tapped. We saw early in the chapter when crossing the same origin barrier that Flash often has capabilities that native JavaScript lacks. Now, when trying to solve the real-time problem, we see that Flash offers us the possibility of TCP socket-based communication, which will provide true continuous connection two-way messaging. So Flash will act as

a binary bridge, making the communication to a socket server and pipe information back and forth to the JavaScript in the page. We note the browser isn't the only one needing assistance, as the socket server will act as a helper to the Web server as well.



As an example of the binary bridge approach, we again use a Flash object helper. Given the following ActionScript code in our file (ajaxtcrflash.as), we see the exposure of a socket method externally.

```
import flash.external.ExternalInterface;
class AjaxTCRFlash{

        static function socket(url, port, callback)
        {
          var socketObj = new XMLSocket();
          socketObj.connect(url, port);
          socketObj.onData = function(input:String) {
                               ExternalInterface.call(callback, input.toString());
                             };
        }
static function main() {
                ExternalInterface.addCallback("socket", null, socket);
        }
}
```

Similar to the cross-domain example earlier in the chapter, we compile this code into a SWF file and take the created SWF file and insert it into the page. We do have to address the various browser differences for inserting and referencing the SWF file, but once it is put in the page, we simply call its externally exposed `socket()` method and signal what the callback is that we want it to populate the page with.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Chapter 10: Socket Time</title>
<script type="text/javascript">

function createSWF()
{
 var swfNode = "";
 if (navigator.plugins && navigator.mimeTypes && navigator.mimeTypes.length)
    swfNode = '<embed type="application/x-shockwave-flash" src=
"http://ajaxref.com/ch10/flash/ajaxtcrflash.swf" width="1" height="1"
id="flashbridge" name="flashbridge" />';
 else {
        swfNode = '<object id="flashbridge" classid=
"clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" width="1" height="1" >';
        swfNode += '<param name="movie" value=
"http://ajaxref.com/ch10/flash/ajaxtcrflash.swf" />';
        swfNode += "</object>";
      }
 document.getElementById("flashHolder").innerHTML = swfNode;
}
function getSWF(movieName)
{
 if (navigator.appName.indexOf("Microsoft")!= -1)
   return window[movieName];
 else
   return document[movieName];
}

function printTime(str)
{
  document.getElementById("responseOutput").innerHTML = str;
}
window.onload = function() {
   createSWF();
   document.getElementById("socketButton").onclick = function(){
                                getSWF("flashbridge").socket("", "7225",
"printTime");}
}
</script>
</head>
<body>
<form action="#">
 <input type="button" value="Socket what time is it? " id="socketButton" />
</form>
<br /><br />
<div id="flashHolder"></div>
<div id="responseOutput"> </div>
</body>
</html>
```

To see real-time communication in your Web browser via Flash, see the example at http://ajaxref.com/ch10/flashsocket.html. It works quite nicely; the only thing you might not like is that the browser status might show a strange communications message:

Server Event Listeners

The WhatWG specification (www.whatwg.org) defines server events to help enable push-style applications. While the specification is still quite new, Opera 9 already contains partial support for this interesting idea, and other browsers are likely to follow. The basic idea is that we include a new tag:

```
<event-source />
```

and set the `src` attribute to a server-side program of interest:

```
<event-source src="servertime.php" id="timeEvent" />
```

We then use JavaScript to bind an event listener to the tag:

```
document.getElementById("timeEvent").addEventListener("update_time",
handleResponse, false);
```

listening for events of particular types and then specifying the callback to handle them.

A complete example that sets up the client side is shown here. Note that we don't bother with direct insertion of the new tag; we just use the DOM to insert it.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Chapter 10 - Opera Server Events</title>
<script src="http://ajaxref.com/ch10/ajaxtcr.js" type="text/javascript"></script>
<script type="text/javascript">
function sendRequest()
{
  var timeEvent = document.createElement("event-source");
  timeEvent.id = "timeEvent";
  timeEvent.setAttribute("src", "opera.php");
  timeEvent.addEventListener("update_time", handleResponse, false);
  document.body.appendChild(timeEvent);
}
function handleResponse(event)
{
 $id("hellodiv").innerHTML = event.data;
}
AjaxTCR.util.event.addWindowLoadEvent(sendRequest);
</script>
```

```
</head>
<body>
<h1>Opera Server Events</h1>
<div id="hellodiv"></div>
</body>
</html>
```

On the server side, we need to pump out events for the browser to receive. We note that we must indicate a new MIME type `application/x-dom-event-stream` for our client updates. We also put the changes in the following form:

```
Event: event-name\n
data:  data-to-send\n\n
```

A very simple program that outputs the time in this event stream format is shown here:

```
<?php
header("Cache-Control: no-cache");
header("Pragma: no-cache");
header("Content-Type: application/x-dom-event-stream");
while (true)
{
 $message = "Hello World at " . date("h:i:s A");
 print "Event: update_time\n";
 print "data: " . $message . "\n\n";
 ob_flush();
 flush();
 sleep(2);
}
?>
```

If you have a browser that supports this style of push, such as Opera 9, give it a whirl at http://ajaxref.com/ch10/opera.html.

NOTE *You may wonder how this idea works communications-wise. Inspection with many browser level monitoring tools will interfere with the communications mechanism, but when we used a raw network capture it appeared that the approach uses an unending HTTP request pattern similar to the endless iframe, at least in the current instantiation in Opera 9.*

The Comet Challenge: Web Chat

If you say anything at all about Comet, you have to include some mention of chat. We implemented a basic chatting system using all the methods previously discussed. You can find a page pointing to each of them at http://ajaxref.com/ch10/chat.html.

Architecturally, chat presents some interesting challenges. For example, when a user types a message, if you wait to get a response back from the server before updating the page, it really seems quite slow to the end user. However, if you directly post the message client side, you face a clock skew problem because your local posts are slightly different than server posted messages. If you opt for posting your own messages locally, you don't need to fetch those from the server; you only want other people's messages. Even monitoring user

liveliness versus posting messages is a bit difficult, with the former requiring that you do replacements of data to keep an up-to-date duplicate-free list of users, while the latter is a continuous appending of data approach to updates. We'll let you dig into the code to see these issues and more; otherwise, you can enjoy chatting as we did in Figure 10-10.

The Comet Impact

Adding Comet-style interaction to your Web site is a potentially dangerous endeavor. The held connection approach, coupled with how many Web servers and application environments are built, can lead to significant scalability problems. For example, PHP doesn't generally let you keep connections open for extended periods of time. This is by design, and the approach leads to the environment's good scalability. Regardless of the application server, you may also see Web servers choking on Comet, consuming and holding memory and processes for each connection. In short, scaling Comet apps can be quite troublesome without careful planning.

Even if you did not face server problems, the approach of held or continuous connections favored by Comet-style applications is quite troubling in light of the browser's two-connection limit we saw in Chapter 6. Of course, you could use another domain name to avoid this, but then you run into the cross-domain concerns. There are ways around this using an iframe with `document.domain` loosening, as we saw in Chapter 7, or using Flash

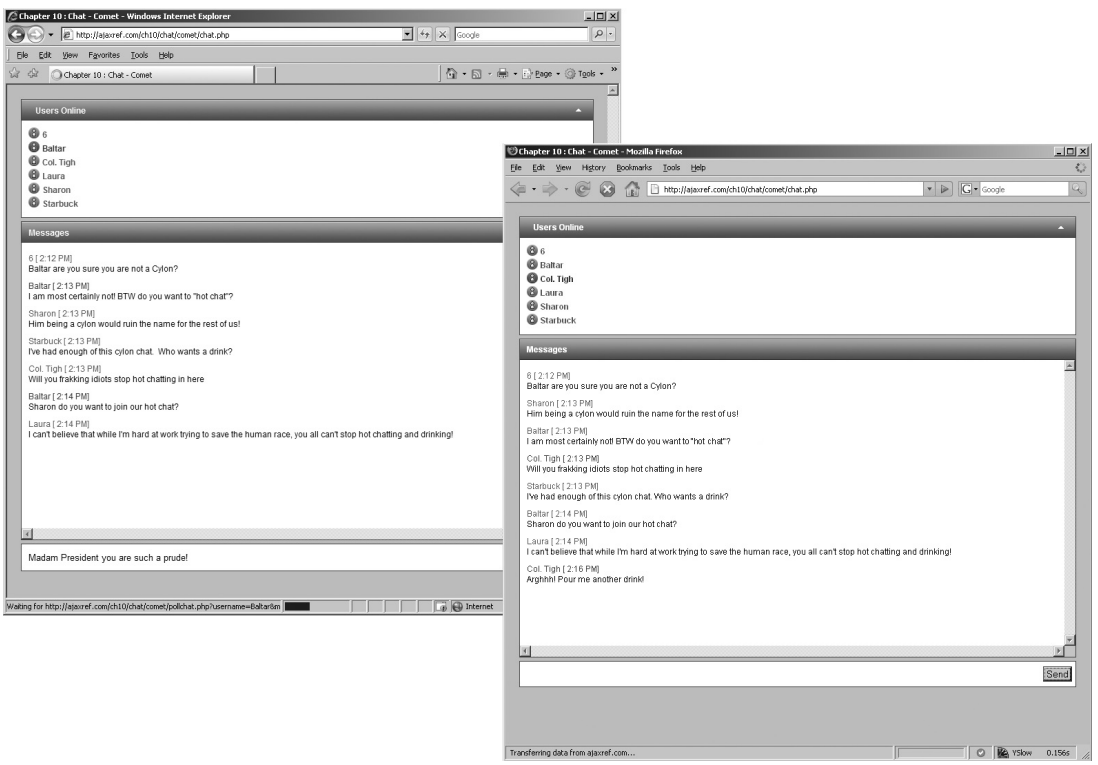


FIGURE 10-10 Chatting Comet style

with a crossdomain.xml file, as we saw in this chapter. Someday, with native XHR support for cross-domain calls, the domain restriction will fall away as we begin to provide multiple DNS entries for our servers, but for now this too is a limitation we must address as well.

The solution to the Comet scale problem comes in two major flavors. The first option is to move to a server-application programming environment architecture pairing that might be more suitable to event-driven long-connection-style coding. One popular platform for this is the Twisted (http://twistedmatrix.com) event-driven networking engine, which is written in Python. The other solution is to use a helper server to offload the long-lived connections but continue to employ the primary environment for normal pages. This is similar to the approach we took in the binary bridge solution using a socket connection.

There is no doubt you can make a push-style application work, but as of yet there is no optimal solution that most agree upon. Those who wish to explore this pattern once again heed the simple warning that as of today, push-style applications will work well in the small but not in the large without some careful planning or even architectural changes.

Going Offline

The final frontier of Web applications using Ajax is going offline. If you could use a Web application on the desktop when you are disconnected from the Internet, say as you fly cross country, and then could later go back online seamlessly, there really is little difference between a desktop application and a Web application. As of late, there's been a bit of envy from Web applications of the desktop richness of offline capabilities, but on the reverse we see desktop apps smarting from the difficulty of distribution and updates that Web applications enjoy. Of course, software applications today rely on the Web to fetch updates and patches to grab this benefit of the network-connected world. It's only fair then that a Web application looks to set up camp on a user's desktop.

What does going offline mean for an Ajax application? What changes will we have to make? First, we need to persist data on the client and rebuild any application state from the persisted data. In the last chapter, we alluded to having such functionality and performed this task in support of history and back button concerns, so we'll start with that. Second, we will need to store resources offline. That might be a bit trickier, and without bleeding edge browsers or extensions like Google Gears, we are out of luck. Finally, we will have to make sure we can work without the network, which will certainly require some careful thinking, interface changes, and extensions like Google Gears. So fasten your seat belts: this last part will get a bit bumpy, but it is well worth the ride.

Client Persistence and Storage

Even if we are always online, we will likely want to persist data between sessions or pages. If this is performed client side, we nearly always turn to cookies. In Chapter 9 we saw that, in support of fixing history and the user's perception of a broken back button, we needed to persist information to make requests or even the responses from previously sent requests. We abstracted the persistence of data away from readers with the library, but here we reveal some of the techniques that can be utilized to persist data. As with many things on the Web, there are many ways to perform the same task, but we stick with the more common solutions to the problem here.

The first and most obvious solution to the persistence challenge are cookies that are easily accessible using JavaScript's `document.cookie` property. While cookies are generally limited to about 4K, we could concatenate data across cookies to provide as much storage as cookies are allowed for a domain.

```
var pieces = Math.floor(value.length/AjaxTCR.storage.DEFAULT_MAX_COOKIE_SIZE + 1);
for (var i=0;i<pieces;i++)
  AjaxTCR.comm.cookie.set(key+i.toString(), value.substring(i*AjaxTCR.storage.
DEFAULT_MAX_COOKIE_SIZE, AjaxTCR.storage.DEFAULT_MAX_COOKIE_SIZE), expires);
```

We have no idea how many cookies were used when reading the data out of a cookie-based storage, but we know the general formula of each piece of the value is key+*piece* where *piece* is an integer starting at zero (for example, savedkey0,savedkey1,savedkey2). So, to read the data out of cookie-style storage, we would use a little algorithm like so:

```
var i=0;
var fullvalue = "";
do {
 var val = AjaxTCR.comm.cookie.get(key+i.toString());
 if (val)
   fullvalue += val;
 i++;
} while(val);

if (fullvalue != "")
  return fullvalue;
```

While the splitting across cookies seems quite expandable, it may be limited to as few as 20 cookies per server, though browsers may allow more. You should assume if you attempt to persist more than 50K with cookie storage you are starting to play with fire.

The second method for persisting data is Internet Explorer's Persistence Behavior. Behavior technology is leftover from the DHTML generation, but don't dismiss this as premillennial technology; it is quite capable. To enable the feature define a style sheet like so:

```
<style type="text/css">
  .storagebin {behavior:url(#default#userData;)}
</style>
```

Then bind it to a `<div>` tag, which serves as a binding container for the storage:

```
<div id="persistThis" class="storagebin"></div>
```

To store things in IE's persistence system, we would then find the `<div>` tag in question using the DOM and use `setAttribute` to define the key-value pair we want to save. However, to commit the data, you must call a `save()` method and pass it a string to reference the data.

```
var persistObj = document.getElementById("persistThis");
persistObj.setAttribute(key,value);
persistObj.save("storageLocker");
```

Note that you can save multiple key-value pairs in a particular store like our "storageLocker" above.

Retrieval is performed similarly. First, fetch the DOM element being used with the persistence behavior. Next, call the `load()` method, passing it the string used as the store (in this case "storageLocker"). Finally, use `getAttribute(key)` to retrieve the value at the passed *key*.

```
var persistObj = document.getElementById("persistThis");
persistObj.load(store);
var value = persistObj.getAttribute(key);
```

The third method for persistence would be using the Flash Player's `SharedObject` and bridging into JavaScript, as we have done for cross-domain requests and socket communication previously in this chapter. This approach is quite appealing because it is transportable between any browser that can use the Flash Player. This means that you can persist data between Internet Explorer and Firefox on the same machine, very powerful and very scary to the privacy minded! Second, we note the scheme typically has a decent size limit of 100KB, though it can be tuned much higher if the user is prompted. Finally, the storage is not known by many users and thus is rarely cleared by them. Of course, it has the obvious downside of requiring Flash in order to work and then relying on the bridge between the two technologies.

The ActionScript code to create a storage system in Flash is quite simple and is shown here in its entirety:

```
import flash.external.ExternalInterface;
class AjaxTCRStorage{
      static var mySharedObject : SharedObject;

      static function add(key, value)
      {
        mySharedObject.data[key] = value;
         mySharedObject.flush();
      }
      static function get(key, value)
      {
        return mySharedObject.data[key];
      }
      static function clear()
      {
        mySharedObject.clear();
      }
      static function remove(key)
      {
        delete mySharedObject.data[key];
      }
      static function getAll()
      {
        return mySharedObject.data;
      }
      static function main()
      {
        mySharedObject=SharedObject.getLocal("AjaxTCRData");
```

```
        ExternalInterface.addCallback("add", null, add);
        ExternalInterface.addCallback("get", null, get);
        ExternalInterface.addCallback("clear", null, clear);
        ExternalInterface.addCallback("remove", null, remove);
        ExternalInterface.addCallback("getAll", null, getAll);
       }
}
```

Similar to the previous examples using a Flash bridge, we can call the various methods in the page directly from JavaScript. First, as before, we have to add the SWF file to the page and then reference it in browser-specific ways. We omit showing this code again since we have seen it twice before already in this chapter. Then, we return a reference to the embedded SWF object.

```
var storageObject = getSWF("flashstorage");
```

To add a value to Flash's storage, we simply call the externally exposed `add()` method, passing the key and value we are interested in storing.

```
storageObject.add("timelord", "the doctor");
```

Retrieving is quite simple as well: just call the external `get()` method on the embedded SWF object and pass it the key of interest and it will return a value if there is one.

```
var val = storageObject.get("timelord");
// returns "the doctor"
```

To further explorer Flash's persistence system, we have provided a demo at http://ajaxref.com/ch10/persistenceflashexplorer.html. You should find it quite interesting and maybe a tad disturbing that you can reference persisted data between browsers using this scheme, as demonstrated in Figure 10-11.

The final solution we present is the native storage mechanism found in Firefox 2-and-up browsers, based upon the WhatWG's (www.whatwg.org) global persistence object. In supporting browsers, you can specify the domain where the storage items are available. For example, `globalStorage[""]` is available to all domains, while `globalStorage["ajaxref.com"]` would be available to ajaxref.com domains and `globalStorage["www.ajaxref.com"]` would just be accessible to that particular domain.

Once you have defined your storage range, you can use `getItem(`*`key`*`)` and `setItem(`*`key`*`,`*`value`*`)` methods on the object like so.

```
var storageObject = globalStorage("ajaxref.com");
storageObject.setItem("secretagent","007");
var value = storageObject.getItem("secretagent");
// returns "007"
```

We summarize each of the storage mechanisms discussed so far in Table 10-3.

We've implemented each of these mechanisms except the Flash approach in the AjaxTCR library, with the library failing back to cookies if another approach is unavailable. The details required to store persistent data regardless of underlying mechanism are as follows.

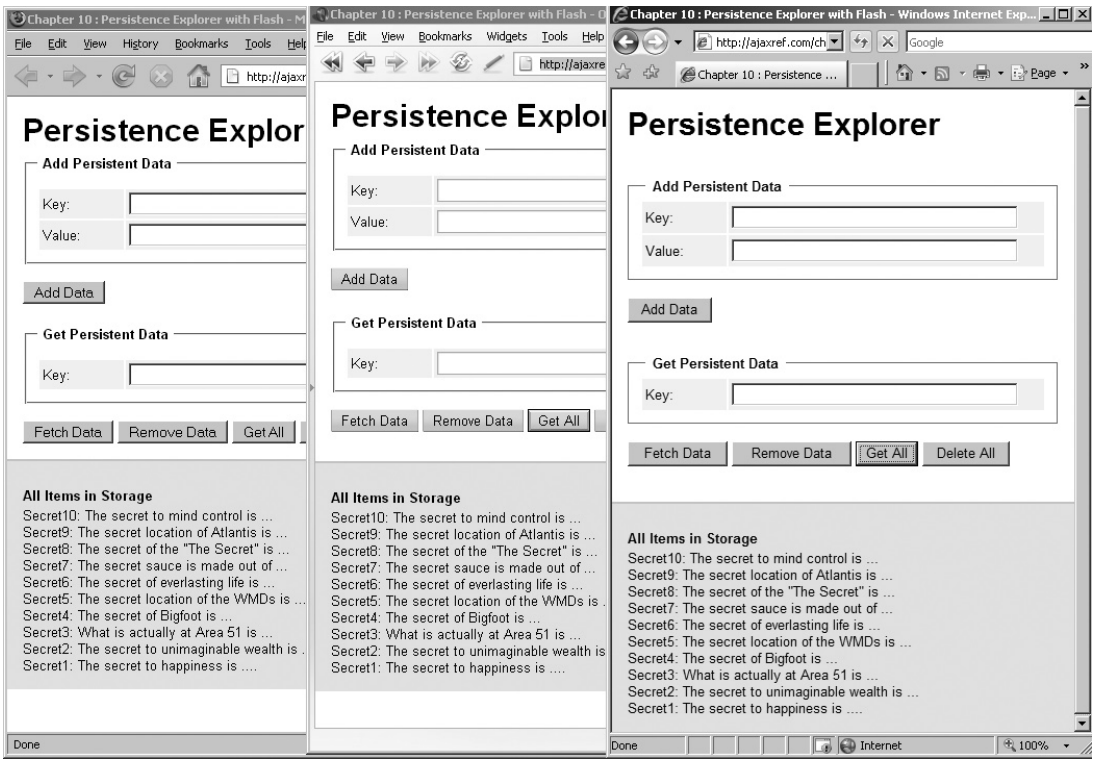


FIGURE 10-11 Sharing persisted data with Flash storage

First, we must initialize the persistence system using the `init()` method, which returns a reference to a persistence object we will use later:

```
var persistObj = AjaxTCR.storage.init();
```

To add a value to the store, we use the `add( )` method, passing it a key and the value we are interested in storing:

```
AjaxTCR.storage.add("way to a mans heart","his stomach")
```

In the case of Internet Explorer we saw we also needed to pass in the persistence object. Thus, `add( )` actually takes that value as well and optionally a storage string value like so:

```
AjaxTCR.storage.add("way to a mans heart","his stomach",persistObj,"default")
```

Because of the differing browser needs, we make the assumption that the `persistObject` must be passed in and that the `store` is optional, though it will default to the value "AjaxTCRStore" when not specified.

To retrieve a value from persistent storage, use the `get()` method, passing it the key and persistence object:

```
var secret = AjaxTCR.storage.add("way to a mans heart",persistObj);
// returned "his stomach"
```

Approach	Description	Comments
Cookies	Stores data in persistent cookies (disk cookies), splitting larger items across a number of cookies to be concatenated together upon retrieval.	Possible in any browser. Subject to cookie cleansing from privacy concerned users. Size and browser limitations. Network impact as the cookie storage would be transmitted every request. Security impact as storage is sent in requests.
Internet Explorer Behaviors	Stores data relative using a DHTML behavior bound to a page element such as a `<div>` tag.	Internet Explorer–specific system. A single page is limited to 64K of persisted data with a whole domain limited to 640K. Without a special cleaning program it may be difficult for users to dump this information.
Flash storage	Uses Flash shared object to store data in browsers that support Flash.	Most bridge between SWF file embedded in page and JavaScript. Shareable between browsers, unlike any other mechanism. By default you should be able to store 100KB of data in this system. It is adjustable with user prompts. Users unlikely to dump persisted data as they are unaware of the storage mechanism.
Native Browser Storage (DOM Storage)WhatWG	A `globalStorage` system is natively available from supporting browsers in JavaScript.	Can be shared across a range of domains and sites. Could be open for abuse. Only implemented in Firefox browsers at this point in time. According to the current spec, a 5MB limit is currently defined, though this may change, particularly if abused.

Table 10-3 Summary of Push-style Communications Approaches

A convenience method of `getAll()` is also provided that returns an array of all items in client persistence.

To remove a value from storage, use the `remove()` method, passing it the key and the persistence object:

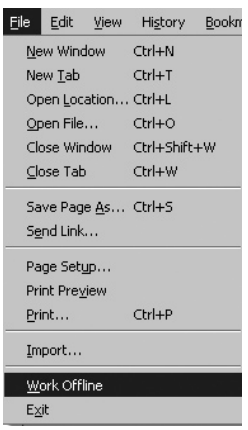
```
AjaxTCR.storage.remove("way to a mans heart",persistObj);
// removed value
```

A similar convenience method, `clear()`, is provided to remove all items from storage. The full syntax of the AjaxTCR storage mechanism is detailed in Table 10-4 and can also be found in Appendix C.

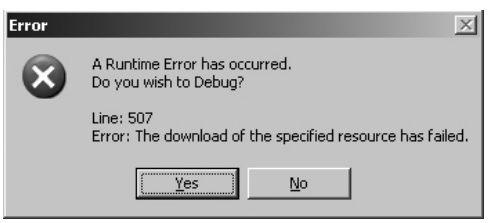
You can try the persistence system using the AjaxTCR library in your browser with our simple explorer program (http://ajaxref.com/ch10/persistenceexplorer.html).

Danger: Offline Ahead!

Just because we have saved some data into our client-side persistent storage, it doesn't necessarily allow us go offline. For example, let's use our Hello World style example. If we go offline in our browser:



and then attempt to make the call, we may raise an exception, depending on the browser. For example, in Firefox 2 we do not seem to have problems as long as we have previously requested the page. However, regardless of a previous request or not, in other browsers like Internet Explorer, you will most likely throw an error when you issue the XHR request in offline mode.



Method	Description	Example
`add(key,value, persistenceObject [, store])`	Stores the value specified as a string at the key specified in the appropriate storage system bound to the persistence object. In the case of Internet Explorer, the `store` parameters may also be passed in otherwise a default value is supplied.	`AjaxTCR.storage.add("fortknox","lots of gold",persistObj)`
`get(key, persistanceObject [,store])`	Retrieves data at the passed key from the storage system related to the `persistenceObject`.	`var treasure = AjaxTCR.storage .get("fortknox", persistObj); alert(treasure); /* shows "lots of gold" */`
`getAll(persistenceObject [,store])`	Retrieves all data from the storage system referenced by the `persistenceObject`.	`var allTreasure = AjaxTCR.storage .getAll(persistObj);`
`init()`	Initializes the data store for holding persisted data. Returns a handle to the persistence object. Persistence system tries to accommodate Firefox and Internet Explorer persistence forms and degrades to cookies if necessary.	`var persistObj = AjaxTCR.storage.init();`
`clear(persistenceObject [,store])`	Clears all the items out of the storage system related to the `persistenceObject`.	`AjaxTCR.storage.clear(persistObj)`
`remove(key, persistanceObject [,store])`	Removes the data from the storage system related to the passed `key`.	`AjaxTCR.storage.remove("fortknox", persistObj); /* no more gold */`

TABLE 10-4 Methods of AjaxTCR.storage

To see for yourself try our simple demo at http://ajaxref.com/ch10/helloworldoffline.html.

Considering our AjaxTCR library supports its own cache, it would seem likely that being offline and accessing a cached response from memory would work, and it does. However, our simple response cache doesn't solve the offline problem, because what would happen when you try to make a new request or post some data in offline mode? Errors, for certain! Of course, if we give the browser something to talk to when it is offline, maybe we can solve that problem too. Enter Google Gears.

Enabling Offline with Google Gears

Google Gears (http://code.google.com/apis/gears/) is an open-source browser extension that provides developers the ability to build Web applications using familiar technologies like JavaScript that can run offline. Google Gears is composed of three components:

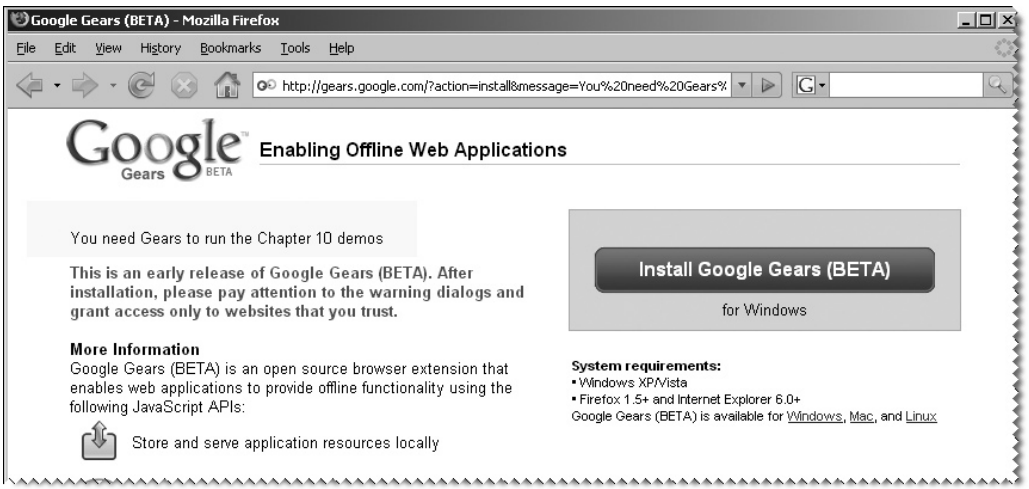
- **A local Web server** Caches and serves the resource components of the Web application (XHTML, CSS, JavaScript, images, and so on) locally in absence of a connection to the Internet
- **A database** Stores data used by our offline applications with an instance of the open source SQLite database (www.sqlite.org/), a fully capable relational database
- **A worker pool extension** Speeds up the processing model of JavaScript, allowing resource-intensive operations to happen asynchronously—in other words, to run in the background.

With these three components installed and enabled, you should be able to perform the necessary functions to go offline.

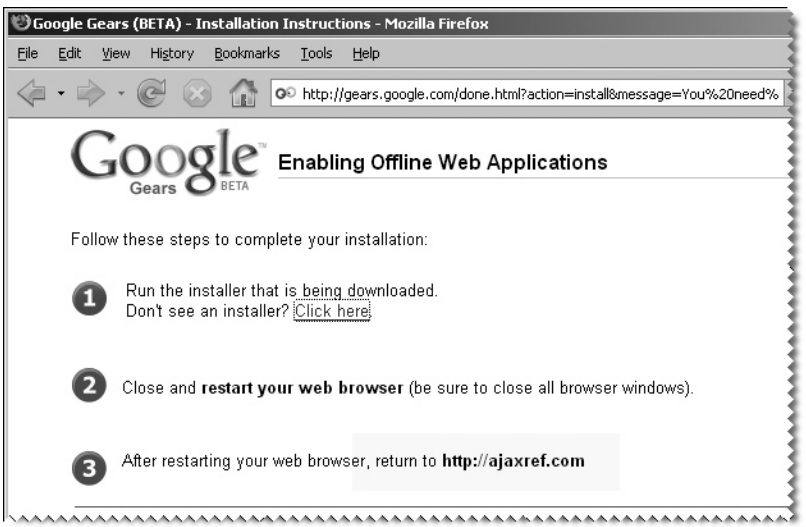
Not everyone is going to have Gears installed, so after you include the Gears library in your code, you will run a simple detection script and bounce them over to the Gears site for installation.

```
<script type="text/javascript" src="gears_init.js"></script>
<script type="text/javascript">
/* global detect for gears */
if (!window.google || !google.gears)
 {
   location.href = "http://gears.google.com/?action=install&message=You need
Gears to run the Ajax: The Complete Reference Chapter 10 offline demos" +
                    "&return=http://ajaxref.com";
 }
</script>
```

Note how Google allows us to provide an installation string to alert the user:



Upon install, it also gives us advice of where to return to:



If everything is installed properly and you start to build your first Gears app, be prepared to be prompted by a browser to allow Gears to run:

User training might be required with such prompts, as otherwise they might think something is amiss.

The first thing you would want to do to go offline is make sure you have the necessary files available for your browser to use. Gears provides an easy way to do this. First, create a special `manifest.json` file indicating the resources you need offline. The file consists of an entries array containing the URLs you would like to have cached:

```
{
   "betaManifestVersion": 1,
   "version": "v1",
   "entries": [
    { "url": "offlinetest.html" },
    { "url": "offlinepage.html" },
    { "url": "images/rufus.jpg" },
    { "url": "scripts/alert.js" },
    { "url": "gears_init.js"}
   ]
}
```

We use relative paths here, but you could use full paths and URL as well.

When the page loads we call our own `initGears()` function, where we create an instance of the local Web server:

```
localServer = google.gears.factory.create("beta.localserver", "1.1");
```

Next, we create a managed store to hold our files:

```
store = localServer.createManagedStore("lockbox");
```

When we desire to save files to the local storage, we first indicate the files we would like to capture:

```
store.manifestUrl = "http://ajaxref.com/ch10/offline/manifest.json";
```

Next we go ahead and grab the files:

```
store.checkForUpdate();
```

As this process may take a while, we start a timer to look every half-second and see if our files are available for offline usage yet:

```
/* check every 500 ms to see if it is all saved or not */
var timerId = window.setInterval(function() {
          if (store.currentVersion)
            {
             window.clearInterval(timerId);
             document.getElementById("responseOutput").innerHTML ="The documents
are now available offline.";
             }
          else if (store.updateStatus == 3)
                document.getElementById("responseOutput").innerHTML =
"Error: " + store.lastErrorMessage;
}, 500);
```

Now that the files are safely stored, if the user were to go offline and attempt to use the files of interest they could do so. If they have not captured the files, they would of course see the expected error message. These scenarios are shown in Figure 10-12.

If for some reason we want to remove the stored data, it is easily done like so:

```
localServer.removeManagedStore("lockbox");
```

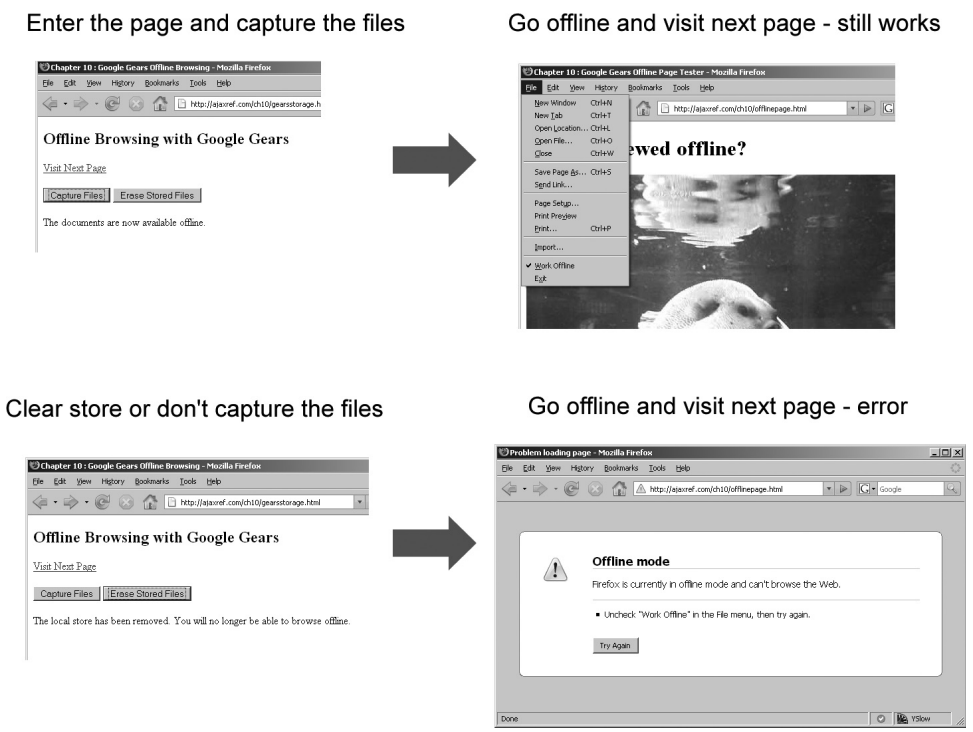


FIGURE 10-12 Offline access: scenarios with Gears

We provide a full example to test the storage mechanism at http://ajaxref.com/ch10/gearsstorage.html, but you can inspect the full code here as well.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<script type="text/javascript" src="gears_init.js"></script>
<title>Chapter 10 : Google Gears Offline Browsing</title>
</head>
<script type="text/javascript">
/* global detect for gears */
if (!window.google || !google.gears)
{
   location.href = "http://gears.google.com/?action=install&message=You need
Gears to run the Ajax: The Complete Reference Chapter 10 offline demos" +
                    "&return=http://ajaxref.com/ch10/gearsstorage.html";
}
var localServer;
var store;
function initGears()
{
 localServer = google.gears.factory.create("beta.localserver", "1.1");
 store = localServer.createManagedStore("lockbox");
}
function createStore()
{
  store.manifestUrl = "http://ajaxref.com/ch10/manifest.json";
  store.checkForUpdate();
  var timerId = window.setInterval(function() {
          if (store.currentVersion)
            {
              window.clearInterval(timerId);
              document.getElementById("responseOutput").innerHTML ="The documents
are now available offline.";
            }
          else if (store.updateStatus == 3)
                document.getElementById("responseOutput").innerHTML =
"Error: " + store.lastErrorMessage;
      }, 500);
}

function removeStore()
{
  localServer.removeManagedStore("lockbox");
  document.getElementById("responseOutput").innerHTML ="The local store has been
removed.  You will no longer be able to browse offline.";
}
window.onload = function(){
initGears();
document.getElementById("captureBtn").onclick = function() {createStore();}
document.getElementById("eraseBtn").onclick = function() {removeStore();}
}
```

```
</script>
<body>
<h2>Offline Browsing with Google Gears </h2>
<a href="offlinepage.html">Visit Next Page</a><br /><br />
<form action="#">
  <input type="button" id="captureBtn" value="Capture Files" />
  <input type="button" id="eraseBtn" value="Erase Stored Files" />
</form>
<br />
<div id="responseOutput"></div>
</body>
</html>
```

Gears also provides an offline database that we can write to. After we initialize Gears, we can create a database with a call like so:

```
db = google.gears.factory.create('beta.database', '1.0');
```

Once we have a handle on our database, we can perform familiar commands upon it. First, we open the database.

```
db.open('database-demo');
```

Next, we execute a SQL statement to create a table to be used offline if it is not there:

```
db.execute('create table if not exists todolist
(todonum int, todo varchar(255))');
```

Later, we can perform normal SQL statements upon the database. For example, here we issue a standard select statement and print out either a message that no data is available in the case no rows are returned, or each row line by line until finished.

```
var todolist = document.getElementById('todolist');
todolist.innerHTML = '';

var rs = db.execute('select * from todolist order by todonum asc');
if (!rs.isValidRow())
 {
   todolist.innerHTML = "<em>No items</em>";
   rs.close()
   return;
  }
while (rs.isValidRow())
  {
   todolist.innerHTML +=  rs.field(0) + ") "+ rs.field(1)  +"<br />";
   rs.next();
  }
 rs.close();
```

It is pretty clear we could build a simple to-do list maker since we have a local database. We see this in Figure 10-13, and you can run the example at http://ajaxref.com/ch10/gearsdb.html.

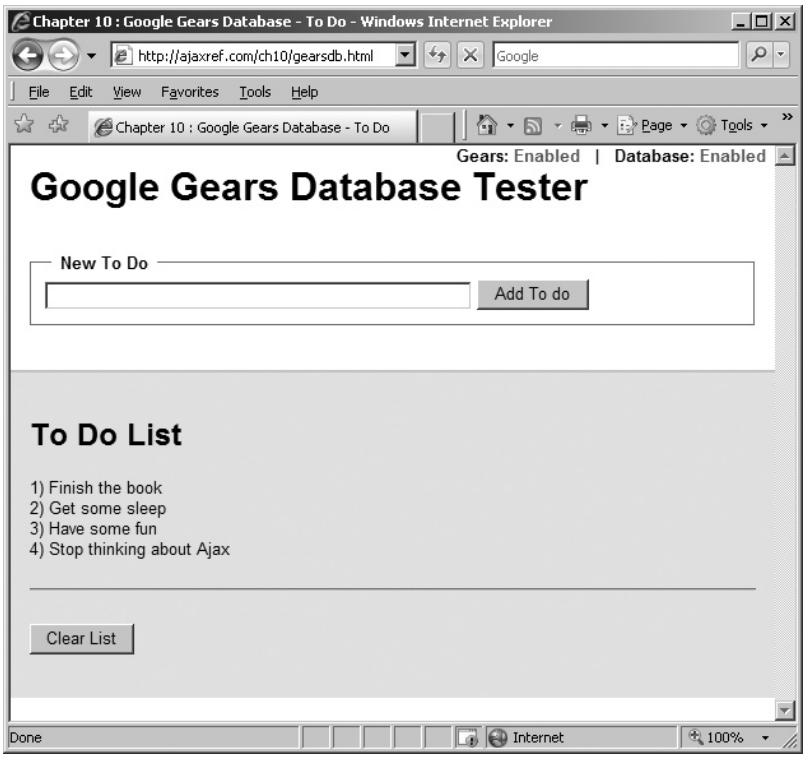


FIGURE 10-13 Gears offline database demo

In Chapter 9, we developed a full blown to-do list application to work with Ajax and degrade nicely even without JavaScript. Here we aim to take this idea and try to make it work offline, but we need to show how we might integrate the two.

It should be clear that the problem that will emerge when we merge these two ideas is how to synchronize data between offline and online modes. For example, you make your to-do items online and then go offline. You may continue to work, but when you come back online you would want your to-do items to be synchronized up. We can opt between two different approaches for handling this, a manual or more automatic approach.

Our thinking to pick one architectural approach over another is driven by how much we want the user to be involved in the process and how connected we think we will be. For example, if we assume that we are mostly connected, we may want more of a manual approach where the user explicitly indicates they want to go offline and bring data down to the local store. We might conversely assume a less connected state and perform tasks with the assumption of being mostly offline and then synching up transparently as we note connectivity being available.

To seamlessly slip between the offline and online mode, we modify the data handling of our sample to-do list application to save the list data in our local Gears database, as well as attempt to commit it online. In our sample to-do application, we assume a connected status and modify our communication to save data locally as well. For example, when we go offline,

our communication will fail so our library will invoke any `onFail` callback we have. We modify our callback so that upon failure we write the change to our local database and set a global dirty flag variable (`g_dirty`) that we use to signal that things are different offline than they are online. If we fail, we also change the visual status to let the user know they are offline.

When requests are going through as normal, we call our `onSuccess` callbacks but we still update our local data store with the same changes made online. Upon every successful request, we have to assume the previous request might not have been successful and check the dirty flag. If it indicates we are out of sync, we call a special sync function to make sure both the local and online application state match. We also update our online status as up when a request goes through. Simple usage of the to-do application on- and offline is shown in Figure 10-14.

The code is a bit involved to present it in paper, so we suggest you trace it carefully online. Entrance to this Gears application can be found at http://ajaxref.com/ch10/gearstodo.

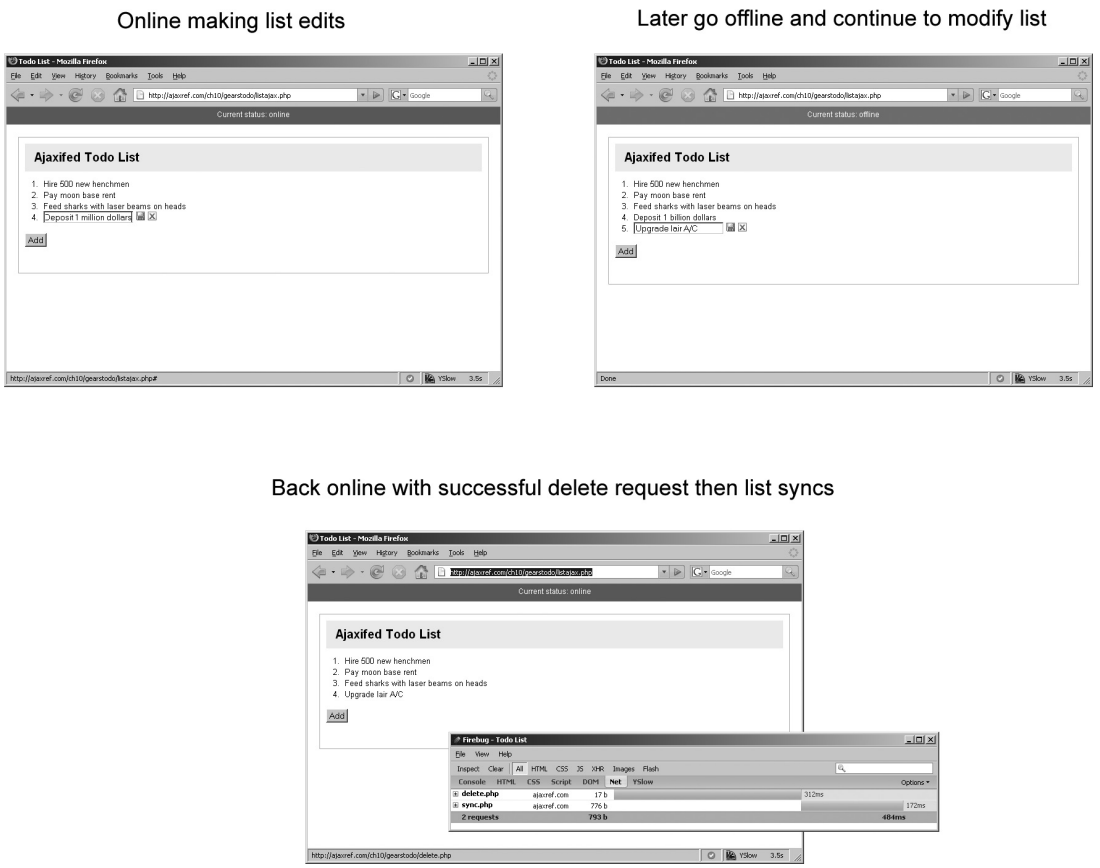


FIGURE 10-14 To-do list, offline and on

Moving between offline and online modes introduces many architectural challenges for a Web application. If the data set is small enough, we can do a mirroring concept, but for larger data sizes this may not be possible. Some applications might need to synchronize automatically, while others make more sense to be synched manually. In all cases, letting the user know the status of the connection and the application state is paramount.

The power that Gears provides is quite exciting and, as we saw with our to-do list, the Web is starting to intrude on the desktop. However, it would seem that if the desktop has an install requirement, Gears doesn't really change much. Simply put, as cool as this approach is, having user's install local proxy software on their systems is not likely over the long haul, especially if we consider that, like everything we have seen in this advanced chapter, the future is browser native!

Emerging Offline Possibilities with Firefox 3

The Firefox 3 browser will likely be out by the time you read this and it has features in it to assist in enabling offline access. First up is the ability to easily detect if you are offline or not by looking at the Boolean value in `navigator.onLine`. Here we toggle a string value based upon this value:

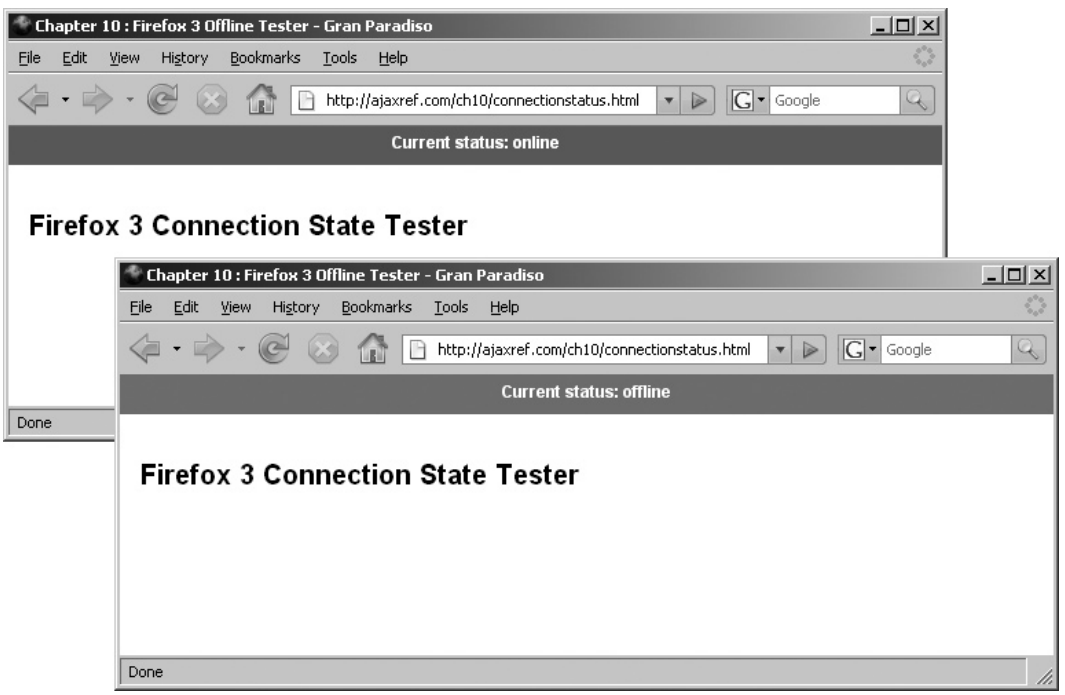
```
var condition = navigator.onLine ? "online" : "offline";
```

However, this won't do us much good unless we can see when the user goes offline and comes back. We certainly could use a timer and check this value every so often, but Firefox 3 also provides an event handler for the events `offline` and `online` that we bind to the body element. The following simple example demonstrates the connection state.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Chapter 10 : Firefox 3 Connection Tester</title>
<link rel="stylesheet" href="http://ajaxref.com/ch10/global.css"
media="screen" />
<style type="text/css">
 #status {height:20px; padding: 4px;
          font-size: 12px;
          color: white;
          text-align:center;}
 #status.online { background-color:green; }
 #status.offline { background-color:red; }
</style>
<script type="text/javascript">
function updateOnlineStatus()
 {
  var condition = navigator.onLine ? "online" : "offline";
  document.getElementById("status").className = condition;
  document.getElementById("state").innerHTML = condition;
 }
window.onload = function () {
```

```
  updateOnlineStatus();
  document.body.addEventListener("offline", updateOnlineStatus, false);
  document.body.addEventListener("online",  updateOnlineStatus, false);
}
</script>
<body>
  <div id="status">Current status: <span id="state"> </span></div>
  <div class="content"><h2>Firefox 3 Offline Tester</div>
</body>
</html>
```

You can see this simple example at http://ajaxref.com/ch10/connectionstatus.html, and it is shown in action here.



In Firefox 3, you can indicate that a resource should be made available for offline consumption simply by setting a `<link>` tag value like so:

```
<link rel="offline-resource"
      href="http://ajaxref.com/ch10/offlineimage.gif" />
```

These items will be loaded after the `onload` event has fired for the page, similar to how prefetching mechanisms work. However, we can programmatically control the process on our own by calling `navigator.offlineResources.add()`, passing it a URL string of what we are interested in saving:

```
navigator.offlineResources.add("http://ajaxref.com/ch10/offlineimage.gif");
```

We can also remove items using `navigator.offlineResources.remove()`, passing it the URL string of what we want to remove from the offline store:

```
navigator.offlineResources.remove("http://ajaxref.com/ch10/offlineimage.gif");
```

For bulk removal use the `clear()` method:

```
navigator.offlineResources.clear(); // no more storage
```

As a list of resources we can look at the length of the `offlineResources`:

```
alert(navigator.offlineResources.length); // How many items
```

We can also look at particular items numerically:

```
alert(navigator.offlineResources.item(1)); // What's at position 1
```

And we can query the list to see if a particular URL is in the list:

```
if (navigator.offlineResources.has("http://ajaxref.com/ch10/secretplans.html"))
  alert("The plans are safely saved offline!");
```


NOTE *The process of saving files for offline use may take some time, and it is possible the user will go offline before it is done. There are interfaces to address this possibility, but at the time of this edition's writing they are still somewhat in flux. Check Firefox's documentation for the latest information on `navigator.pendingOfflineLoads` and the load events associated with it.*

An example similar to the Gears offline storage demo but using Firefox 3's native offline support is shown here:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Chapter 10 : Firefox 3 Offline Browsing</title>
</head>
<script type="text/javascript">
var prefix = "http://ajaxref.com/ch10/";
var filesToStore =   [ "offlinestorage.html" , "offlinepage.html" ,
"images/rufus.jpg" , "scripts/alert.js" ];
function createStore()
{
 var i;
 for (var i=0; i < filesToStore.length; i++)
 {
  try {
    navigator.offlineResources.add(prefix+filesToStore[i]);
  } catch (e) { };
 }
}
function removeStore()
{
  navigator.offlineResources.clear();
```

```
  document.getElementById("responseOutput").innerHTML ="The local store has been
removed.  You may no longer be able to browse offline.";
}

window.onload = function(){
 document.getElementById('captureBtn').onclick = function() {createStore();}
 document.getElementById('eraseBtn').onclick = function() {removeStore();}
}
</script>
<body>
<h2>Offline Browsing with Firefox 3</h2>
<a href="offlinepage.html">Visit Next Page</a><br /><br />
<form action="#">
  <input type="button" id="captureBtn" value="Capture Files" />
  <input type="button" id="eraseBtn"   value="Erase Stored Files" />
</form>
<br />
<div id="responseOutput"></div>
</body>
</html>
```

We do not show the operation visually, as it is the same as the previous Gears example, but you can try it for yourself in a Firefox 3 or better browser by visiting http://ajaxref.com/ch10/offlinestorage.html.

If Firefox 3 supported a local database, it would seem we could pretty much forego the use of systems like Gears almost altogether. Interestingly, with `globalStorage` we might be able to hack something together to do just that. However we might not need to with SQLite built in to Firefox; maybe this will be exposed to browser JavaScript someday soon.

Regardless of the exact details of using Gears or native browser facilities, with the emergence of offline support and all the other facilities we have seen in this chapter and earlier in the book, it would appear the dream of viewing the browser as a development platform has finally arrived—only about a decade later than when Netscape and others first proposed it!

Summary

In our final pages we took some time exploring some of the yet-to-be determined areas of Ajax and client-side Web development. First we saw that given the same origin policy uncertainty of Ajax, the role of direct client consumption of various Web Services using XHRs is not a certainty at this point in time. Workarounds using `<script>` tags, while commonplace, do have their concerns and lack a degree of control, which makes server proxies necessary. Ajax isn't really built yet for direct Web Services. Similarly, Ajax is intrinsically a pull-style technology. Using various long polling techniques or bridging via binaries can provide the real time update, but it is clunky. Comet isn't on the developer's lips just yet because the pattern and supporting technology is still in its early stages of development, even compared to Ajax. However, upcoming changes in browsers such as server-side event listeners show that big changes might be coming soon. Finally, offline access on the desktop presents the final frontier for Ajax—while still quite raw, once we get there, the difference between Web application and desktop application melts away. However, Ajax developers might get more than they bargained for: if users apply desktop presentation and quality expectations of Web software to our Ajax applications, we might find we have quite a lot of interface work to do.