

What is Ajax?

From 2001 to 2005, the World Wide Web went through a tremendous growth spurt in terms of the technologies and methodologies being used to bring this once-static medium to life.

Ajax Is Born

In February 2005, Jesse **James Garrett** of Adaptive Path, LLC published an online article entitled, “[Ajax: A New Approach to Web Applications](http://www.adaptivepath.com/publications/essays/archives/000385.php)” (still available at www.adaptivepath.com/publications/essays/archives/000385.php). In this essay, Garrett explained how he believed web applications were closing the gap between the Web and traditional desktop applications. He cited new technologies and several of the Google projects as examples of how traditionally desktop-based user interaction models were now being used on the Web. Then came the two sentences that would ignite a firestorm of interest, excitement, and controversy:

Google Suggest and Google Maps are two examples of a new approach to web applications that we at Adaptive Path have been calling Ajax. The name is shorthand for **Asynchronous JavaScript + XML**, and it represents a fundamental shift in what’s possible on the Web.

From that point forward, a tidal wave of Ajax articles, code samples, and debates began popping up all over the Web. Developers blogged about it, technology magazines wrote about it, and companies began hitching their products to it. But to understand what Ajax is, you first must understand how the evolution of several web technologies led to its development.

The Evolution of the Web

When Tim Berners-Lee crafted the first proposal for the World Wide Web in 1990, the idea was fairly simple: to create a “web” of interconnected information using **hypertext** and **Uniform Resource Identifiers** (URIs). The ability to link disparate documents from all around the world held huge potential for scholarly endeavors, where people would be able to access referenced material almost instantly. Indeed, the first version of the **Hypertext Markup Language (HTML)** featured little more than formatting and linking commands, a platform not for building rich interactive software but rather for sharing the kinds of textual and illustrative information that dominated the late age of print.

As the Web evolved, businesses saw potential in the ability to distribute information about products and services to the masses. But a small company called **Netscape** would soon be ready to push the evolution of the Web forward at a much faster pace.

JAVASCRIPT

Netscape Navigator was the first successful mainstream web browser, and as such, moved web technologies along quickly. However, Netscape often was ridiculed by standards organizations for implementing new technologies and extensions to existing technologies before the standards were in place. One such technology was JavaScript.

Originally named LiveScript, JavaScript was created by Brendan Eich of Netscape and included in version 2.0 of the browser (released in 1995). For the first time, developers were able to affect how a web page could interact with the user. Instead of making constant trips to the server and back for simple tasks such as data validation, it became possible to transfer this small bit of processing to the browser.

This ability was very important at a time when most Internet users were connected through a 28.8 Kbps modem, turning every request to the server into a waiting game. Minimizing the number of times that the user had to wait for a response was the first major step toward the Ajax approach.

Frames

The original version of HTML intended for every document to be standalone, and it wasn't until HTML 4.0 that frames were officially introduced. The idea that the display of a web page could be split up into several documents was a radical one, and controversy brewed as Netscape chose to implement the feature before the HTML 4.0 standard was completed. Netscape Navigator 2.0 was the first browser to support frames and JavaScript together. This turned out to be a major step in the evolution of Ajax.

When the browser wars of the late 1990s began between Microsoft and Netscape, both JavaScript and frames became formalized. As more features were added to both technologies, creative developers began experimenting using the two together. Because a frame represented a completely separate request to the server, the ability to control a frame and its contents with JavaScript opened the door to some exciting possibilities.

The Hidden Frame Technique

- As developers began to understand how to manipulate frames, a new technique emerged to facilitate client-server communication.
- The hidden frame technique involved setting up a frameset where one frame was set to a width or height of 0 pixels, its sole purpose being to initiate communication with the server.
- The hidden frame would contain an HTML form with specific form fields that could be dynamically filled out by JavaScript and submitted back to the server.
- When the frame returned, it would call another JavaScript function to notify the calling page that data had been returned.

- The hidden frame technique represented the [first asynchronous request/response model](#) for web applications.
 - While this was the first Ajax communication model, another technological advance was just around the corner.
-

Dynamic HTML and the DOM

- In [1996](#), the Web was still mainly a static world.
 - Although JavaScript and the hidden frame technique livened up the user interaction, there was still no way to change the display of a page without reloading it, aside from changing the values contained within form fields. [Then came Internet Explorer 4.0.](#)
 - At this point, Internet Explorer had caught up with the technology of market leader Netscape Navigator and even one-upped it in one important respect through the [introduction of Dynamic HTML \(DHTML\).](#)
 - Although still in the development phase, DHTML represented a significant step forward from the days of static web pages, enabling developers to alter any part of a loaded page by using JavaScript.
 - DHTML never made it to a standards body, although Microsoft's influence would be felt strongly with the [introduction of the Document Object Model \(DOM\)](#) as the centerpiece of the standards effort.
 - Unlike DHTML, which sought only to modify sections of a web page, the [DOM had a more ambitious purpose: to provide a structure for an entire web page.](#)
 - The manipulation of that structure would then allow DHTML-like modifications to the page.
 - This was the next step towards Ajax.
-

Iframes

Although the hidden frame technique became incredibly popular, it had a downside — one had to plan ahead of time and write a `frameset` anticipating the usage of hidden frames.

When the element was introduced as an official part HTML 4.0 in 1997, it represented another significant step in the evolution of the Web.

Iframes

```
<!DOCTYPE html>
<html>
<body>
<h2>HTML Iframes</h2>
<iframe src="gb.html" height="200" width="300" title="Iframe Example"></iframe>
</body></html>
```

FRAMESET

```
<!DOCTYPE html>
<html>
<head>
<meta charset = "utf-8">
<title>A First Program in JavaScript</title>
<script type="text/javascript" src="embed.js"></script>
</head>
<frameset cols="50%, 50%">
  <FRAMESET rows="50%, 50%">
    <frame src="E://My Programs//parseInt.html"/>
    <frame src="E://My Programs//parsetoday.html"/>
  </frameset>
<FRAMESET rows="50%, 50%">
  <frame src="E://My Programs//parseInt.html"/>
  <frame src="E://My Programs//parsetoday.html"/>
</frameset>
</frameset>
<body></body></html>
```

- Instead of defining framesets, developers could place **iframes** anywhere on a page.
 - This enabled developers to forego framesets altogether and simply place invisible iframes (through the use of CSS) on a page to enable client-server communication.
 - And when the **DOM was finally implemented in Internet Explorer 5 and Netscape 6**, it introduced the ability to dynamically create iframes on the fly, meaning that a JavaScript function could be used to create an iframe, make a request, and get the response — all without including any additional HTML in a page.
 - This led to the next generation of the hidden frame technique: the hidden iframe technique.
-

XMLHttpRequest

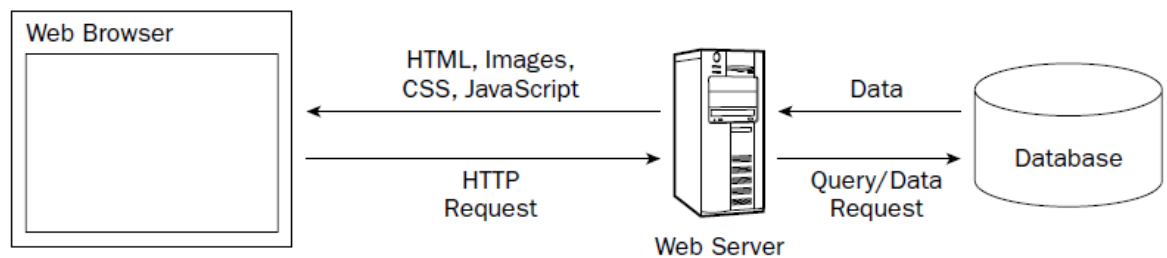
- The browser developers at Microsoft must have realized the popularity of the hidden frame technique and the newer hidden iframe technique, because they decided to provide developers with a better tool for client-server interaction.
 - That tool came in the form of an **ActiveX object** called XMLHttpRequest, introduced in 2001.
 - One of the Microsoft extensions to JavaScript allowed the creation of ActiveX controls, Microsoft's proprietary programming objects.
 - When **Microsoft began supporting XML through a library called MSXML**, the XMLHttpRequest object was included. Although it carried the XML name, this object was more than just another way of manipulating XML data.
 - Developers had access to HTTP status codes and headers, as well as any data returned from the server.
 - That data might be structured XML, pre-formatted swaths of HTML, serialized JavaScript objects, or data in any other format desired by the developer.
 - Instead of using hidden frames or iframes, it was now possible to access the server programmatically using pure JavaScript, independent of the page load/reload cycle.
 - The XMLHttpRequest object became a tremendous hit for Internet Explorer developers.
 - With popularity mounting, developers at the open source Mozilla project began their own port of XMLHttpRequest. Instead of allowing access to ActiveX, the **Mozilla developers replicated the object's principal methods and properties in a native browser object, XMLHttpRequest**.
-

The real Ajax

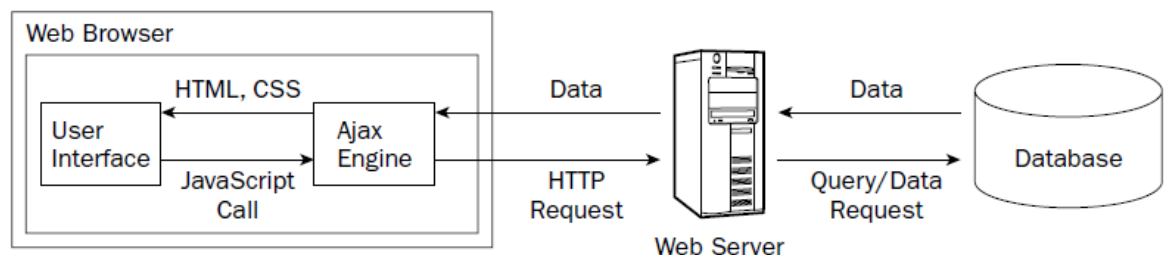
- Despite the frequently asked questions attached to the end of Garrett's essay, some confusion still exists as to what Ajax really is.
- Put simply, Ajax is nothing more than an approach to web interaction.

- This approach involves transmitting only a small amount of information to and from the server in order to give the user the most responsive experience possible.
- Instead of the traditional web application model where the browser itself is responsible for initiating requests to, and processing requests from, the web server, the Ajax model provides an intermediate layer — what **Garrett calls an Ajax engine**— to handle this communication.
- An Ajax engine is really just a JavaScript object or function that is called whenever information needs to be requested from the server.
- Instead of the traditional model of providing a link to another resource (such as another web page), each link makes a call to the Ajax engine, which schedules and executes the request.
- The request is done asynchronously, meaning that code execution doesn't wait for a response before continuing.
- When the Ajax engine receives the server response, it goes into action, often parsing the data and making several changes to the user interface based on the information it was provided.
- Because this process involves transferring less information than the traditional web application model, user interface updates are faster, and the user is able to do his or her work more quickly.

Traditional Web Application Model



Ajax Web Application Model



Ajax Principles

Michael Mahemoff, a software developer and usability expert, identified several key principles of good Ajax applications that are worth repeating:

Minimal traffic:

Ajax applications should send and receive as little information as possible to and from the server. In short, Ajax can minimize the amount of traffic between the client and the server. Making sure that your Ajax application doesn't send and receive unnecessary information adds to its robustness.

No surprises:

Ajax applications typically introduce different user interaction models than traditional web applications. As opposed to the web standard of click-and-wait, some Ajax applications use other user interface paradigms such as drag-and-drop or double-clicking. No matter what user interaction model you choose, be consistent so that the user knows what to do next.

Established conventions:

Don't waste time inventing new user interaction models that your users will be unfamiliar with. Borrow heavily from traditional web applications and desktop applications, so there is a minimal learning curve.

No distractions:

Avoid unnecessary and distracting page elements such as looping animations and blinking page sections. Such gimmicks distract the user from what he or she is trying to accomplish.

Accessibility:

Consider who your primary and secondary users will be and how they most likely will access your Ajax application. Don't program yourself into a corner so that an unexpected new audience will be completely locked out. Will your users be using older browsers or special software? Make sure you know ahead of time and plan for it.

Avoid entire page downloads:

All server communication after the initial page download should be managed by the Ajax engine. Don't ruin the user experience by downloading small amounts of data in one place but reloading the entire page in others.

User first:

Design the Ajax application with the users in mind before anything else. Try to make the common use cases easy to accomplish and don't be caught up with how you're going to fit in advertising or cool effects.

The common thread in all these principles is usability. Ajax is, primarily, about enhancing the web experience for your users; the technology behind it is merely a means to that end.

Technologies behind Ajax

- ❑ **HTML/XHTML:** Primary content representation languages
- ❑ **CSS:** Provides stylistic formatting to XHTML 6 Chapter 1 04_109496 ch01.qxd 2/5/07 6:47 PM Page 6
www.it-ebooks.info
- ❑ **DOM:** Dynamic updating of a loaded page
- ❑ **XML:** Data exchange format
- ❑ **XSLT:** Transforms XML into XHTML (styled by CSS)
- ❑ **XMLHttp:** Primary communication broker
- ❑ **JavaScript:** Scripting language used to program an Ajax engine

In reality, all these technologies are available to be used in Ajax solutions, but only three are required: HTML/XHTML, DOM, and JavaScript.

XHTML is obviously necessary for the displaying of information, while the DOM is necessary to change portions of an XHTML page without reloading it.

The last part, JavaScript, is necessary to initiate the client-server communication and manipulate the DOM to update the web page.

Who Is Using Ajax?

- Google Suggest
 - Gmail
 - Google Maps
 - A9
 - Yahoo! News
 - Bitflux Blog
-

Ajax and Web 2.0

Shortly after the term Ajax was coined, another term began popping up. Web 2.0 was originally the name of a conference held by O'Reilly Media and CMP Media in late 2005. After that, the term Web 2.0 took on a life of its own and began popping up all over the Internet in descriptions of how the Web had changed.

Web 2.0 includes:

1. The Web as services, not software
2. The group mentality of the Web — users encouraged to participate (as with tagging, blogging, networking, and so on)
3. Separation of data and presentation — data can be represented in any number of ways and combined with any other data sources (called mashups)
4. Richer, more responsive user experience

Chapter -2

Ajax Basics

The driving force behind Ajax is the interaction between the client (web browser) and the server.

HTTP Primer

Central to a good grasp of Ajax techniques is [hypertext transmission protocol \(HTTP\)](#), the protocol to transmit web pages, images, and other types of files over the Internet to your web browser and back.

Whenever you type a URL into the browser, an “[http://](#)” is prepended to the address, indicating that you will be using HTTP to access the information at the given location.

[HTTP consists of two parts](#): a **request** and a **response**. When you type a URL in a web browser, the browser creates and sends a request on your behalf.

This request contains the URL that you typed in as well as some information about the browser itself.

The server receives this request and sends back a response. The response contains information about the request as well as the data located at the URL (if any). It's up to the browser to interpret the response and display the web page (or other resource).

HTTP Requests

The format of an HTTP request is:

```
<request-line>  
<headers>  
<blank line>  
[<request-body>]
```

In an HTTP request, the first line must be a request line indicating the type of request, the resource to access, and the version of HTTP being used. Next, a section of headers indicate additional information that may be of use to the server. After the headers is a blank line, which can optionally be followed by additional data (called the body).

There are a large number of request types defined in HTTP, but the two of interest to Ajax developers are **GET and POST**. Anytime you type a URL in a web browser, the browser sends a GET request to the server for that URL, which basically tells the server to get the resource and send it back.

Here's what a GET request for www.wrox.com might look like:

```
GET / HTTP/1.1  
Host: www.wrox.com  
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)  
           Gecko/20050225 Firefox/1.0.1  
Connection: Keep-Alive
```

The **first part** of the request line specifies this as a GET request.

The **second part** of that line is a forward slash (/), indicating that the request is for the root of the domain.

The **last part** of the request line specifies to use HTTP version 1.1 (the alternative is 1.0).

And where is the request sent? That's where the second line comes in.

The **second line** is the first header in the request, Host. The Host header indicates the target of the request. Combining Host with the forward slash from the first line tells the server that the request is for www.wrox.com/. (The Host header is a requirement of HTTP 1.1; the older version 1.0 didn't require it.) The **third line** contains the **User-Agent header**, which is accessible to both server- and client-side scripts and is the cornerstone of most browser-detection logic.

The **last line** is the Connection header, which is typically set to **Keep-Alive for browser operations** (it can also be set to other values, but that's beyond the scope of this book). Note that there is a single blank line after this last header. Even though there is no request body, the blank line is required.

If you were to request a page under the www.wrox.com domain, such as <http://www.wrox.com/books>, the request would look like this:

```
GET /books/ HTTP/1.1
Host: www.wrox.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
          Gecko/20050225 Firefox/1.0.1
Connection: Keep-Alive
```

Note that only the first line changed, and it contains only the part that comes after `www.wrox.com` in the URL. Sending parameters for a GET request requires that the extra information be appended to the URL itself. The format looks like this:

```
URL?name1=value1&name2=value2&...&nameN=valueN
```

This information, called a *query string*, is duplicated in the request line of the HTTP request, as follows:

```
GET /books/?name=Professional%20Ajax HTTP/1.1
Host: www.wrox.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
          Gecko/20050225 Firefox/1.0.1
Connection: Keep-Alive
```

Note that the text “Professional Ajax” had to be encoded, replacing the space with `%20`, in order to send it as a parameter to the URL.

This is called **URL encoding** and is used in many parts of HTTP.

The **name-value pairs are separated with an ampersand**. Most server-side technologies will decode the request body automatically and provide access to these values in some sort of logical manner. Of course, it is up to the server to decide what to do with this data.

The **POST request, on the other hand, provides additional information to the server in the request body**. Typically, when you fill out an online form and submit it, that data is being sent through a POST request.

Here’s what a typical POST request looks like:

```
POST / HTTP/1.1
Host: www.wrox.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.6)
          Gecko/20050225 Firefox/1.0.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 40
Connection: Keep-Alive
```

```
name=Professional%20Ajax&publisher=Wiley
```

You should note a few differences between a POST request and a GET request.

First, the request line begins with “POST” instead of “GET,” indicating the type of request. You’ll notice that the Host and User-Agent headers are still there, along with two new ones. The **Content-Type header** indicates how the request body is encoded. Browsers always encode post data as `application/x-www-`

formurlencoded, which is the MIME type for simple URL encoding. The Content-Length header indicates the byte length of the request body. After the Connection header and the blank line is the request body.

As with most browser POST requests, this is made up of simple name-value pairs, where name is Professional Ajax and publisher is Wiley. You may recognize that this format is the same as that of query string parameters on URLs.

As mentioned previously, there are other HTTP request types, but they follow the same basic format as GET and POST.

The next step is to take a look at what the server sends back in response to an HTTP request. For security purposes, GET requests should be used to retrieve information only.

If data needs to be added, updated, or deleted, a POST request should be used.

HTTP Response

The format of an HTTP response, which is very similar to that of a request, is:

```
<status-line>
<headers>
<blank line>
[<response-body>]
```

As you can see, the only real difference in a response is that the first line contains status information instead of request information. The status line tells you about the requested resource by providing a status code. Here's a sample HTTP response:

```
HTTP/1.1 200 OK
Date: Sat, 31 Dec 2005 23:59:59 GMT
Content-Type: text/html; charset=ISO-8859-1
Content-Length: 122

<html>
  <head>
    <title>Wrox Homepage</title>
  </head>
  <body>
    <!-- body goes here -->
  </body>
</html>
```

the status line gives an HTTP status code of 200 and a message of OK. The status line always contains the status code and the corresponding short message so that there isn't any confusion. The most common status codes are:

❑ **200 (OK):** The resource was found and all is well.

❑ **304 (NOT MODIFIED):** The resource has not been modified since the last request. This is used most often for browser cache mechanisms.

❑ **401 (UNAUTHORIZED):** The client is not authorized to access the resource. Often, this will cause the browser to ask for a user name and password to log in to the server.

❑ **403 (FORBIDDEN):** The client failed to gain authorization. This typically happens if you fail to log in with a correct user name and password after a 401.

❑ **404 (NOT FOUND):** The resource does not exist at the given location.

Typically, the server will return a Date header indicating the date and time that the response was generated. (Servers typically also return some information about themselves, although this is not required.)

The next two headers should look familiar as well, as they are the same **Content-Type** and **Content-Length** headers used in POST requests. In this case, the Content-Type header specifies the MIME type for HTML (text/html) with an encoding of ISO-8859-1 (which is standard for the United States English resources). The body of the response simply contains the HTML source of the requested resource (although it could also contain plain text or binary data for other types of resources). It is this data that the browser displays to the user.

Ajax Communication Techniques

Now that you understand the basics of how HTTP communication works, it's time to look into enacting such communication from within a web page.

The Hidden Frame Technique

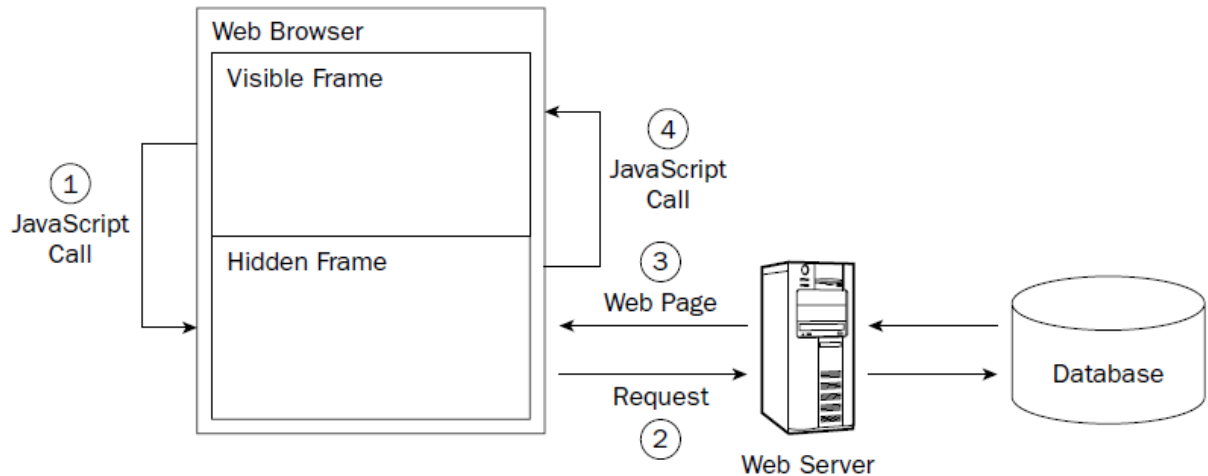
With the introduction of HTML frames, the hidden frame technique was born. The basic idea behind this technique is to create a frameset that has a hidden frame that is used for client-server communication. You can **hide a frame by setting its width or height to 0 pixels**, effectively removing it from the display. Although some early browsers (such as Netscape 4) couldn't fully hide frames, often leaving thick borders, this technique still gained popularity among developers.

The Pattern

The hidden frame technique follows a very specific, four-step pattern (see Figure).

The **first step** always begins with the visible frame, where the user is interacting with a web page. Naturally, the user is unaware that there is a hidden frame (in modern browsers, it is not rendered) and goes about interacting with the page as one typically would. At some point, the user performs an action that requires additional data from the server. When this happens, the first step in the process occurs: a JavaScript function call is made to the hidden frame. This call can be as simple as redirecting the hidden

frame to another page or as complicated as posting form data. Regardless of the intricacy of the function, the result is the **second step** in the process: a request made to the server.



The **third step** in the pattern is a response received from the server. Because you are dealing with frames, this response must be another web page. This web page must contain the data requested from the server as well as some JavaScript to transfer that data to the visible frame.

Typically, this is done by assigning an **onload** event handler in the returned web page that calls a function in the visible frame after it has been fully loaded (this is the fourth step). With the data now in the visible frame, it is up to that frame to decide what to do with the data.

Hidden Frame GET Requests

Create a database and table in mysql .Install Xampp

With the database table all set up, it's time to move on to the HTML code. To use the hidden frame technique, you must start with an HTML frameset, such as this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<html>
  <head>
    <title>Hidden Frame GET Example</title>
  </head>
  <frameset rows="100%,0" style="border: 0px">
    <frame name="displayFrame" src="DataDisplay.php" noresize="noresize" />
    <frame name="hiddenFrame" src="about:blank" noresize="noresize" />
  </frameset>
</html>
```

The important part of this code is the rows attribute of the element.

By setting it to **100%,0**, browsers know not to display the body of the second frame, whose name is hiddenFrame.

Next, the style attribute is used to set the border to 0, ensuring that there isn't a visible border around each frame.

The final important step in the frameset declaration is to set the noresize attributes on each frame so that the user can't inadvertently resize the frames and see what's in the hidden one; the contents of the hidden frame are never meant to be part of the visible interface.

Next up is the page to request and display the customer data (DataDisplay.jsp). This is a relatively simple page, consisting of a textbox to enter the customer ID, a button to execute the request, and a element to display the retrieved customer information:

```
<p>Enter customer ID number to retrieve information:</p>
<p>Customer ID: <input type="text" id="txtCustomerId" value="" /></p>
<p><input type="button" value="Get Customer Info"
        onclick="requestCustomerInfo()" /></p>
<div id="divCustomerInfo"></div>
```

You'll notice that the button calls a function named requestCustomerInfo(), which interacts with the hidden frame to retrieve information.

It simply takes the value in the textbox and adds it to the query string of GetCustomerData.jsp, creating a URL in the form of GetCustomerData.jsp?id=23. This URL is then assigned to the hidden frame. Here's the function:

```
function requestCustomerInfo() {
    var sId = document.getElementById("txtCustomerId").value;
    top.frames["hiddenFrame"].location = "GetCustomerData.php?id=" + sId;
}
```

To get a reference to the hidden frame, you first need to access the topmost window of the browser using the top object.

That object has a frames array, within which you can find the hidden frame. Since each frame is just another window object, you can set its location to the desired URL.

That's all it takes to request the information. Note that because the request is a GET (passing information in the query string), it makes the request very easy.

In addition to the requestCustomerInfo() function, you'll need another function to display the customer information after it is received. This function, displayCustomerInfo(), will be called by the hidden frame when it returns with data. The sole argument is a string containing the customer data to be displayed:

```
function displayCustomerInfo(sText) {
    var divCustomerInfo = document.getElementById("divCustomerInfo");
    divCustomerInfo.innerHTML = sText;
}
```

In this function, the first line retrieves a reference to the element that will display the data. In the second line, the customer info string (sText) is assigned into the innerHTML property of the element. Using innerHTML makes it possible to embed HTML into the string for formatting purposes. This completes the code for the main display page.

Now it's time to create the server-side logic: `GetCustomerData.jsp`

The basic code for `GetCustomerData.jsp` is a very basic HTML page with JSP code in two places

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="java.sql.*" %>
<!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>
        <title>Get Customer Data</title>
        <script type="text/javascript">
            window.onload = function () {
                var divInfoToReturn = document.getElementById("divInfoToReturn");
                top.frames["displayFrame"].displayCustomerInfo(divInfoToReturn.innerHTML);
            };
        ]]]&gt;
    &lt;/script&gt;
&lt;/head&gt;
&lt;body&gt;
&lt;%!
protected String getCustomerData(int id){
    try {
        Class.forName("com.mysql.jdbc.Driver").newInstance();</pre>
</div>
```



```
//database information
String dbservername = "localhost";
String dbname = "ProAjax";
String username = "root";
String password = "admin";

String url = "jdbc:mysql://" + dbservername + "/" + dbname + "?user=" + username + "&password="
+ password;
```

```
//create database connection
Connection conn = DriverManager.getConnection(url);
```

```
//execute query
String sql = "Select * from Customers where CustomerId=" + id;
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
boolean found = rs.next();
StringBuffer buffer = new StringBuffer();
```

```
//if there was a match...
```

```
if (found) {
```

```
    buffer.append(rs.getString("Name"));
    buffer.append("<br />");
    buffer.append(rs.getString("Address"));
    buffer.append("<br />");
    buffer.append(rs.getString("City"));
    buffer.append("<br />");
    buffer.append(rs.getString("State"));
    buffer.append("<br />");
```

```

        buffer.append(rs.getString("Zip"));

        buffer.append("<br /><br />");

        buffer.append("Phone: " + rs.getString("Phone"));

        buffer.append("<br /><a href=\"mailto:\"");

        buffer.append(rs.getString("Email"));

        buffer.append("\">");

        buffer.append(rs.getString("Email"));

        buffer.append("</a>");

    } else {

        buffer.append("Customer with ID ");

        buffer.append(id);

        buffer.append(" could not be found.");

    }

    rs.close();

    conn.close();

    return buffer.toString();

} catch (Exception ex){

    return "An error occurred while trying to get customer info.";

}

}

%>

<%

String strId = request.getParameter("id");

int id = -1;

String message = "";

```

```

try {
    id = Integer.parseInt(strId);
    message = getCustomerData(id);
} catch (Exception ex) {
    message = "Invalid customer ID.";
}
%>
<div id="divInfoToReturn"><%= message %></div>
</body>
</html>

```

In this page, the first JSP block will contain the logic to retrieve customer data. The second JSP block outputs the variable (=message), containing customer data, into a <div>. It is from this <div> that the data is read and sent to the display frame. To do so, create a JavaScript function that is called when the page has loaded completely:

```

window.onload = function () {
    var divInfoToReturn = document.getElementById("divInfoToReturn");
    top.frames["displayFrame"].displayCustomerInfo(divInfoToReturn.innerHTML);
};

```

- This function is assigned directly to the `window.onload` event handler.
- It first retrieves a reference to the <div/> that contains the customer information.
- Then, it accesses the display frame using the `top.frames` array and calls the `displayCustomerInfo()` function defined earlier, passing in the `innerHTML` of the <div/>.
- That's all the JavaScript it takes to send the information where it belongs. But how does the information get there in the first place?
- Now when `<%=message>` is output into the <div/>, it will contain the appropriate information.
- The `onload` event handler reads that data out and sends it back up to the display frame.

Hidden Frame POST Requests

- A POST request is typically sent when data needs to be sent to the server as opposed to a GET, which merely requests data from the server.
- Although GET requests can send extra data through the query string, some browsers can handle only up to 512KB of query string information.
- A POST request, on the other hand, can send up to 2GB of information, making it ideal for most uses.

[Refer programs for Hidden Frame Post and Get in GCR](#)

Hidden iFrames

[Refer programs for Hidden Frame Post and Get in GCR](#)


The next generation of behind-the-scenes client-server communication was to make use of [iframes \(short for inline frames\)](#), which were introduced in HTML 4.0.

Basically, an iframe is the same as a frame with the exception that it can be placed inside of a non-frameset HTML page, effectively allowing any part of a page to become a frame. Iframe can even be created on the fly in JavaScript.

Because iframes can be used and accessed in the same way as regular frames, they are [ideal for Ajax communication](#).

There are [two ways](#) to take advantage of iframes. The easiest way is to simply embed an iframe inside of your page and use that as the hidden frame to make requests. Doing this would change the first example display page to:

[\(iFrame creation : way 1 \)](#)



```
<p>Enter customer ID number to retrieve information:</p>
<p>Customer ID: <input type="text" id="txtCustomerId" value="" /></p>
<p><input type="button" value="Get Customer Info"
      onclick="requestCustomerInfo()" /></p>
<div id="divCustomerInfo"></div>
<iframe src="about:blank" name="hiddenFrame" style="display: none"></iframe>
```

- Note that the iframe has its style attribute set to “display:none”; this effectively hides it from view. Since the name of the iframe is hiddenFrame, all the JavaScript code in this page will continue to work as before.
- There is, however, one small change that is necessary to the GetCustomerData.jsp page.
- The JavaScript function in that page previously looked for the displayCustomerInfo() function in the frame named displayFrame.
- If you use this technique, there is no frame with that name, so you must update the code to use parent instead:

```

window.onload = function () {
    var divInfoToReturn = document.getElementById("divInfoToReturn");
    parent.displayCustomerInfo(divInfoToReturn.innerHTML);
};

```

When accessed inside of an iframe, the parent object points to the window (or frame) in which the iframe resides.

The second way to use hidden iframes is to create them dynamically using JavaScript.

[\(iFrame creation : way 2 \)](#)

```

function createIFrame() {
    var oIFrameElement = document.createElement("iframe");
    oIFrameElement.style.display = "none";
    oIFrameElement.name = "hiddenFrame";
    oIFrameElement.id = "hiddenFrame";
    document.body.appendChild(oIFrameElement);

    //more code
}

```

The last line of this code is very important because it adds the iframe to the document structure; an iframe that isn't added to the document can't perform requests. Also note that both the name and id attributes are set to hiddenFrame.

This is necessary because some browsers access the new frame by its name and some by its id attribute.

Next, define a global variable to hold a reference to the frame object. Note that the frame object for an iframe element isn't what is returned from createElement(). In order to get this object, you must look into the frames collection. This is what will be stored in the global variable:

```

oIFrame = frames["hiddenFrame"];

```

```

function requestCustomerInfo() {
    if (!oIFrame) {
        createIFrame();
        setTimeout(requestCustomerInfo, 10);
        return;
    }

    var sId = document.getElementById("txtCustomerId").value;
    oIFrame.location = "GetCustomerData.php?id=" + sId;
}

```

With these changes, the function now checks to see if oIFrame is null or not. If it is, it calls createIFrame() and then [sets a timeout to run the function again in 10 milliseconds](#). This is necessary because only Internet Explorer recognizes the inserted iframe immediately; most other browsers take a couple of milliseconds to recognize it and allow requests to be sent.

Hidden iFrame POST Requests

To accomplish a POST request using hidden iframes, the basic approach is to load a page that contains a form into the hidden frame, populate that form with data, and then submit the form. When the visible form (the one you are actually typing into) is submitted, you need to cancel that submission and forward the information to the hidden frame.

```

function checkIFrame() {
    if (!oIFrame) {
        createIFrame();
    }
    setTimeout(function () {
        oIFrame.location = "ProxyForm.php";
    }, 10);
}

```

This function, checkIFrame(), first checks to see if the hidden iframe has been created. If not, createIFrame() is called. Then, a timeout is set before setting the location of the iframe to ProxyForm.php, which is the hidden form page. Because this function may be called several times, it's important that this page be loaded each time the form is submitted.

```

<!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>
  <title>Proxy Form</title>
  <script type="text/javascript">
    //

        window.onload = function () {
            parent.formReady();
        }

    //]]&gt;
  &lt;/script&gt;
&lt;/head&gt;
&lt;body&gt;
  &lt;form method="post" action="#"&gt;
  &lt;/form&gt;
&lt;/body&gt;
&lt;/html&gt;
</pre>
</div>
<div data-bbox="111 468 889 539" data-label="Text">
<p>As you can see, the body of this page contains only an empty form and the head contains only an onload event handler. When the page is loaded, it calls <code>parent.formReady()</code> to let the main page know that it is ready to accept a request. The <code>formReady()</code> function is contained in the main page itself and looks like this:</p>
</div>
<div data-bbox="114 552 688 789" data-label="Text">
<pre>
function formReady() {
    var oForm = document.forms[0];
    var oHiddenForm = oIFrame.document.forms[0];

    for (var i=0 ; i &lt; oForm.elements.length; i++) {
        var oField = oForm.elements[i];

        switch (oField.type) {

            //ignore buttons
            case "button":
            case "submit":
            case "reset":
</pre>
</div>
```

```

        break;

        //checkboxes/radio buttons - only return the value if the control is checked.
        case "checkbox":
        case "radio":
            if (!oField.checked) {
                break;
            }

        //text/hidden/password all return the value
        case "text":
        case "hidden":
        case "password":
            createInputField(oHiddenForm, oField.name, oField.value);
            break;

        default:
            switch(oField.tagName.toLowerCase()) {
                case "select":
                    createInputField(oHiddenForm, oField.name,
                                      oField.options[oField.selectedIndex].value);
                    break;
                default:
                    createInputField(oHiddenForm, oField.name, oField.value);
            }
        }
    }

    oHiddenForm.action = oForm.action;
    oHiddenForm.submit();
};

```

- The first step in this function is to get a reference to the form in the hidden iframe, which you can do by [accessing the document.forms collection](#) of that frame.
- Because there is only one form on the page, you can safely get the first form in the collection ([at index 0](#)); this is stored in oHiddenForm.
- Following that, a reference to the form on the main page is saved into oForm. [Next, a for loop iterates through the form elements on the main page \(using the elements collection\).](#)
- For each form element, a new hidden input element is created in the hidden frame using the createInputField() function (defined in a moment). Since there can be many different types of form elements, this code takes into account the different ways that values are stored.
- [Buttons are ignored](#), since their values are usually unimportant; [checkboxes and radio buttons are included only if they are checked](#); [textboxes are always included](#); [select boxes are given the correct value for the selected option](#). The function to create the fields is defined as follows:


```
function createInputField(oHiddenForm, sName, sValue) {
    oHidden = oIFrame.document.createElement("input");
    oHidden.type = "hidden";
    oHidden.name = sName;
    oHidden.value = sValue;
    oHiddenForm.appendChild(oHidden);
}
```

```
<form method="post" action="SaveCustomer.php"
    onsubmit="checkIFrame();return false">
    <p>Enter customer information to be saved:</p>
    <p>Customer Name: <input type="text" name="txtName" value="" /><br />
    Address: <input type="text" name="txtAddress" value="" /><br />
    City: <input type="text" name="txtCity" value="" /><br />
    State: <input type="text" name="txtState" value="" /><br />
    Zip Code: <input type="text" name="txtZipCode" value="" /><br />
    Phone: <input type="text" name="txtPhone" value="" /><br />
    E-mail: <input type="text" name="txtEmail" value="" /></p>
    <p><input type="submit" value="Save Customer Info" /></p>
</form>
<div id="divStatus"></div>
```

By returning false in this way, you are preventing the default behavior of the form (to submit itself to the server).

Instead, the checkIFrame() method is called and the process of submitting to the hidden iframe begins. With this complete, you can now use this example the same way as the hidden frame POST example; the SaveCustomer.jsp page handles the data and calls saveResult() in the main page when completed.

[Advantages and Disadvantages of Hidden Frames](#)

Be careful, because iframes don't always store browser history.

Whereas IE always stores the history of iframes, Firefox does so only if the iframe was defined using HTML (that is, not created dynamically using JavaScript). Safari never stores browser history for iframes, regardless of how they are included in the page.

One of the biggest arguments for using hidden frames is that you can maintain the browser history and thus enable users to still use the Back and Forward buttons in the browser.

Because the browser doesn't know that a hidden frame is, in fact, hidden, it keeps track of all the requests made through it.

Whereas the main page of an Ajax application may not change, the changes in the hidden frame mean that the Back and Forward buttons will move through the history of that frame instead of the main page. This technique is used in both Gmail and Google Maps for this very reason.

Hidden frames do have some disadvantages.

For one, (1) **you cannot make requests outside of your own domain**. Due to security restrictions in browsers, JavaScript can only interact with frames that are from the same domain. Even a page from a subdomain (such as p2p.wrox.com instead of www.wrox.com) can't be accessed.

Another downside of hidden frames is that there is (2) **very little information about what's going on behind the scenes**. You are completely reliant on the proper page being returned.

The examples in this section all had the same problem: (3) **if the hidden frame page fails to load, there is no notification to the user that a problem has occurred; the main page will continue to wait until the appropriate JavaScript function is called**.

You may be able to provide some comfort to a user by setting a timeout for a long period of time, maybe five minutes, and displaying a message if the page hasn't loaded by then, but that's just a workaround.

The main problem is that (4) **you don't have enough information about the HTTP request that is happening behind the scenes**.

XMLHttpRequests (XHR)

When Microsoft Internet Explorer 5.0 introduced a rudimentary level of XML support, an **ActiveX library called MSXML** was also introduced.

One of the objects provided in this library quickly became very popular: XMLHttpRequest.

The XMLHttpRequest object was created to enable developers to initiate HTTP requests from anywhere in an application.

These requests were intended to return XML, so the XMLHttpRequest object provided an easy way to access this information in the form of an XML document. Since it was an ActiveX control, XMLHttpRequest could be used not only in web pages but also in any Windows-based desktop application;

however, its popularity on the Web has far outpaced its popularity for desktop applications.

Picking up on that popularity, Mozilla duplicated the XMLHttpRequest functionality for use in its browsers, such as Firefox. They created a native JavaScript object, XMLHttpRequest, which closely mimicked the behavior of Microsoft's XMLHttpRequest object. Shortly thereafter, both the Safari (as of version 1.2) and Opera (version 7.6) browsers duplicated Mozilla's implementation. Microsoft even went back and created their own native XMLHttpRequest object for Internet Explorer 7.

Today, all four browsers support a native XMLHttpRequest object, commonly referred to as XHR.

Creating an XHR Object

The first step to using an XHR object is, obviously, to create one. Because Microsoft's implementation prior to Internet Explorer 7 is an ActiveX control, you must use the proprietary ActiveXObject class in JavaScript, passing in the XHR control's signature:

```
var oXHR = new ActiveXObject("Microsoft.XMLHttp");
```

This line creates the first version of the XHR object (the one shipped with IE 5.0). The problem is that there have been several new versions released with each subsequent release of the MSXML library. Each release brings with it better stability and speed, so you want to make sure that you are always using the most recent version available on the user's machine. The signatures are:

- ☐ Microsoft.XMLHttp
 - ☐ MSXML2.XMLHttp
 - ☐ MSXML2.XMLHttp.3.0
 - ☐ MSXML2.XMLHttp.4.0
 - ☐ MSXML2.XMLHttp.5.0
 - ☐ MSXML2.XMLHttp.6.0
-

Windows Vista ships with version 6.0, which is the preferable version to use if able.

Unfortunately, the only way to determine which version to use is to try to create each one. Because this is an ActiveX control, any failure to create an object will throw an error, which means that you must enclose each attempt within a try...catch block. The end result is a function such as this:

```
function createXHR() {  
    var aVersions = [ "MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0"];  
  
    for (var i = 0; i < aVersions.length; i++) {  
        try {  
            var oXHR = new ActiveXObject(aVersions[i]);  
            return oXHR;  
        } catch (oError) {  
            //Do nothing  
        }  
    }  
    throw new Error("MSXML is not installed.");  
}
```

Now this function first checks to see if an XMLHttpRequest class is defined (by using the typeof operator). If XMLHttpRequest is present, it is used to create the XHR object; otherwise, it checks to see if the ActiveXObject class is present and, if so, goes through the same process of creating an XHR object for IE 6 and below. If both of these tests fail, an error is thrown.

Using XHR

- ❑ **Request Type:** A string indicating the request type to be made — typically, GET or POST (these are the only ones currently supported by all browsers)
- ❑ **URL:** A string indicating the URL to send the request to
- ❑ **Async:** A Boolean value indicating whether the request should be made asynchronously

The last argument, `async`, is very important because it controls how JavaScript executes the request. When set to `true`, the request is sent asynchronously, and JavaScript code execution continues without waiting for the response; you must use an event handler to watch for the response to the request. If `async` is set to `false`, the request is sent synchronously, and JavaScript waits for a response from the server before continuing code execution. That means if the response takes a long time, the user cannot interact with the browser until the response has completed. For this reason, best practices around the development of Ajax applications favor the use of asynchronous requests for routine data retrieval, with synchronous requests reserved for short messages sent to and from the server.

To make an asynchronous GET request to a file such as `info.txt`, you would start by doing this:

```
var oXHR = XMLHttpRequest.createRequest();  
oXHR.open("get", "info.txt", true);
```

Next, you need to define an `onreadystatechange` event handler. The XHR object has a property called `readyState` that changes as the request goes through and the response is received. There are five possible values for `readyState`:

- ❑ **0 (Uninitialized):** The object has been created but the `open()` method hasn't been called.
- ❑ **1 (Loading):** The `open()` method has been called but the request hasn't been sent.
- ❑ **2 (Loaded):** The request has been sent.
- ❑ **3 (Interactive):** A partial response has been received.
- ❑ **4 (Completed):** All data has been received and the connection has been closed.

Every time the `readyState` property changes from one value to another, the `readystatechange` event fires and the `onreadystatechange` event handler is called.

The `onreadystatechange` event handler is typically defined as:

```
var oXHR = zXmlHttp.createRequest();
oXHR.open("get", "info.txt", true);
oXHR.onreadystatechange = function () {
    if (oXHR.readyState == 4) {
        alert("Got response.");
    }
};
```

The last step is to call the `send()` method, which actually sends the request. This method accepts a single argument, which is a string for the request body.

```
oXHR.send(null);
```

That's it! The request has been sent and when the response is received, an alert will be displayed. But just showing a message that the request has been received isn't very useful. The true power of XHR is that you have access to the returned data, the response status, and the response headers.

To retrieve the data returned from the request, you can use the `responseText` or `responseXML` properties. The `responseText` property returns a string containing the response body, whereas the `responseXML` property is an XML document object used only if the data returned has a content type of `text/xml`.

```
var sData = oXHR.responseText;
```

e.g, `info.txt` didn't exist, then the `responseText` would contain the server's 404 message.

The `status` property contains the HTTP status code sent in the response, and `statusText` contains the text description of the status (such as "OK" or "Not Found"). Using these two properties, you can make sure that the data you've received is actually the data you want or tell the user why the data wasn't retrieved:

```
if (oXHR.status == 200) {
    alert("Data returned is: " + oXHR.responseText);
} else {
    alert("An error occurred: " + oXHR.statusText);
}
```

Generally, you should always ensure that the status of a response is 200, indicating that the request was completely successful. The `readyState` property is set to 4 even if a server error occurred, so just checking that is not enough. In this example, the `responseText` property is shown only if the status is 200; otherwise, the error message is displayed.

```
if (oXHR.status == 200 || oXHR.status == 304) {
    alert("Data returned is: " + oXHR.responseText);

} else {
    alert("An error occurred: " + oXHR.statusText);
}
```

If a 304 is returned, the `responseText` and `responseXML` properties will still contain the correct data. The only difference is that data comes from the browser's cache instead of from the server.

As mentioned previously, it's also possible to access the response headers. You can retrieve a specific header value using the `getResponseHeader()` method and passing in the name of the header that you want to retrieve. One of the most useful response headers is `Content-Type`, which tells you the type of data being sent:

```
var sContentType = oXHR.getResponseHeader("Content-Type");
if (sContentType == "text/xml") {
    alert("XML content received.");
} else if (sContentType == "text/plain") {
    alert("Plain text content received.");
} else {
    alert("Unexpected content received.");
}
```

To see all the headers from the server

```
var sHeaders = oXHR.getAllResponseHeaders();
var aHeaders = sHeaders.split(/\r?\n/);

for (var i=0; i < aHeaders.length; i++) {
    alert(aHeaders[i]);
}
```

In order to set our own heard before the request is made use the below statement

```
oXHR.setRequestHeader("myheader", "myvalue");
```

Synchronous Requests

```
var oXHR = zXmlHttp.createRequest();
oXHR.open("get", "info.txt", false);
oXHR.send(null);

if (oXHR.status == 200) {
    alert("Data returned is: " + oXHR.responseText);
} else {
    alert("An error occurred: " + oXHR.statusText);
}
```

Sending the request synchronously (setting the third argument of `open()` to `false`) enables you to start evaluating the response immediately after the call to `send()`. This can be useful if you want the user interaction to wait for a response or if you're expecting to receive only a very small amount of data (for

example, less than 1K). In the case of average or larger amounts of data, it's best to use an asynchronous call. There is a chance that a synchronous call will never return if u have infinite loop.

XHR GET Requests

The requestCustomerInfo() function previously created a hidden iframe but now must be changed to use XHR:

```
var oXHR = zXmlHttp.createRequest();
oXHR.open("get", "GetCustomerData.php?id=" + sId, true);
oXHR.onreadystatechange = function () {
    if (oXHR.readyState == 4) {
        if (oXHR.status == 200 || oXHR.status == 304) {
            displayCustomerInfo(oXHR.responseText);
        } else {
            displayCustomerInfo("An error occurred: " + oXHR.statusText);
        }
    }
};
oXHR.send(null);
}
```

There are several differences between this and the hidden frame/iframe example.

First, (1)[no JavaScript code is required](#) outside of the main page.

This is important because any time you need to keep code in two different places there is the possibility of creating incompatibilities; in the frame-based examples, you relied on separate scripts in the display page and the hidden frames to communicate with one another.

The second difference is that it's much easier to tell (2)[if there was a problem executing the request](#). In previous examples, there was no mechanism by which you could identify and respond to a server error in the request process. Using XHR, all server errors are revealed to you as a developer, enabling you to pass along meaningful error feedback to the user. In many ways, XHR is a more elegant solution than hidden frames for in-page HTTP requests.

Refer coding for GET and POST example of XHR requests.

Advantages and Disadvantages of XHR

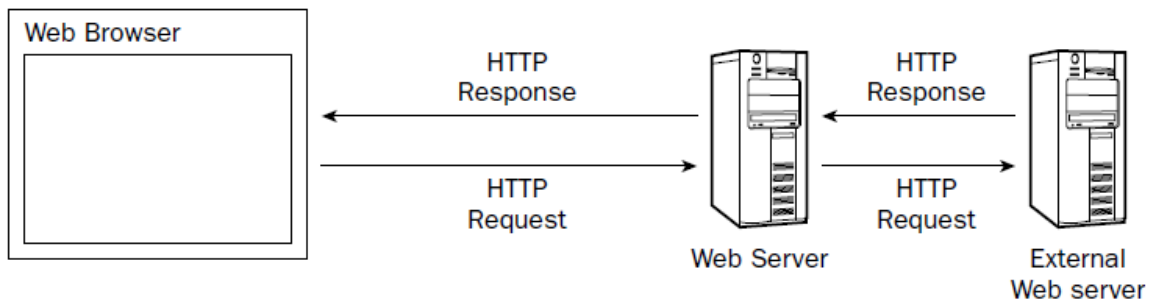
The downside is that, unlike hidden frames, there is no browser history record of the calls that were made. The Back and Forward buttons do not tie in to XHR requests, so you have effectively cut off their use. It is for this reason that many Ajax applications use a mixture of XHR and hidden frames to make a truly usable interface.

Another disadvantage, which applies to Internet Explorer 6 and earlier only, is that you depend on ActiveX controls being enabled. If the user has your page set up in a particular security zone that doesn't allow ActiveX controls, the code cannot access the XHR object. In that case, you may have to default to using hidden frames.

It is also worth noting that XHR has the same restrictions as hidden frames when it comes to crossdomain communication.

Even XMLHttpRequest was designed for making ad hoc requests from JavaScript, it still doesn't break the cross-domain scripting rules. **An XHR object is still only allowed to access resources from the same domain.** If you need to access a URL located in a different origin, you must create a server-side proxy to handle the communication.

Server-Side Proxies



Dynamic Script Loading

A little-known and little-utilized Ajax technique is called dynamic script loading. The concept is simple: create a new `<script/>` element and assign a JavaScript file to its `src` attribute to load JavaScript that isn't initially written into the page. The beginnings of this technique could be seen way back when Internet Explorer 4.0 and Netscape Navigator 4.0 ruled the web browser market. At that time, developers learned that they could use the `document.write()` method to write out a `<script/>` tag. The caveat was that this had to be done before the page was completely loaded. With the advent of the DOM, the concept could be taken to a completely new level.

The basic technique behind dynamic script loading is very easy, just create a `<script/>` element using the DOM `createElement()` method and add it to the page:

```
var oScript = document.createElement("script");
oScript.type = "text/javascript";
oScript.src = "/path/to/my.js";
document.body.appendChild(oScript);
```

[Refer complete example in GCR]

Advantages and Disadvantages of dynamic script loading

Though dynamic script loading is a quick and easy way to establish client-server communication, it does have some drawbacks.

❑ For one, there is(1) no feedback as to what is going on once the communication is initiated. If, for example, the file you are accessing doesn't exist, there is no way for you to receive a 404 error from the server. Your site or application may sit, waiting, because the callback function was never called.

❑ Also, you (2) can't send a POST request using this technique, only a GET, which limits the amount of data that you can send. This could also be a security issue: make sure that you don't send confidential information such as passwords using dynamic script loading, as this information can easily be picked up from the query string.

Dynamic script loading does offer a couple of advantages over other techniques as well.

❑ First, just like using images, it is possible to access files on other servers. This can be very powerful if you are working in a multidomain environment.

❑ Further, dynamic script loading offers the ability to execute an arbitrary amount of JavaScript as the result of server-side calculations. You aren't limited to simply one callback function; use as many as necessary to achieve the desired results.
