Ajax
Asynchronous JavaScript + XML

Mark Andreessen, Netscape, 1995: "MS Windows will be reduced to a poorly debugged set of device drivers running under Netscape Navigator, with desktop-style applications running inside the browser". This did not happen until 10 years later (true/false?)

Asynchronous *JavaScript* + *XML*

- Ajax isn't a technology.
- It's really several technologies. Ajax incorporates:
  - standards-based presentation using **XHTML**;
  - **CSS**, dynamically manipulated using JavaScript;
  - dynamic display and interaction using the Document Object Model (**DOM**). Web page exposed as DOM object;
  - data interchange using **XML** (nowadays **JSON**);
  - asynchronous data retrieval using **XMLHttpRequest**, a JavaScript object, a.k.a "Web remoting";
  - **JavaScript** binding everything together;
  - Server no longer performs display logic, only business logic.
- Acronym originated by **Jesse James Garrett** in **2005**:

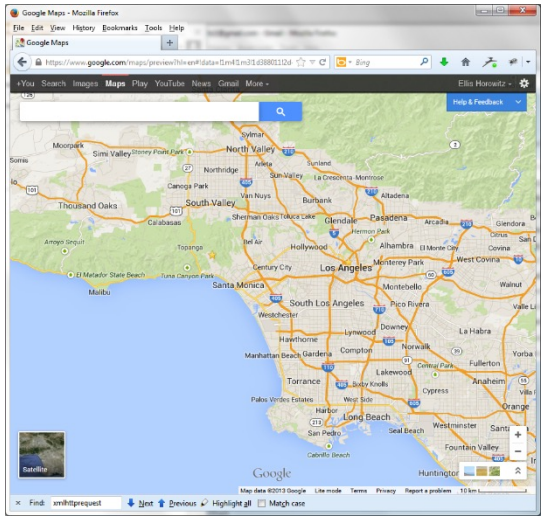
  https://immagic.com/eLibrary/ARCHIVES/GENERAL/ADTVPATH/A050218G.pdf

Some History and Browsers Supporting Ajax

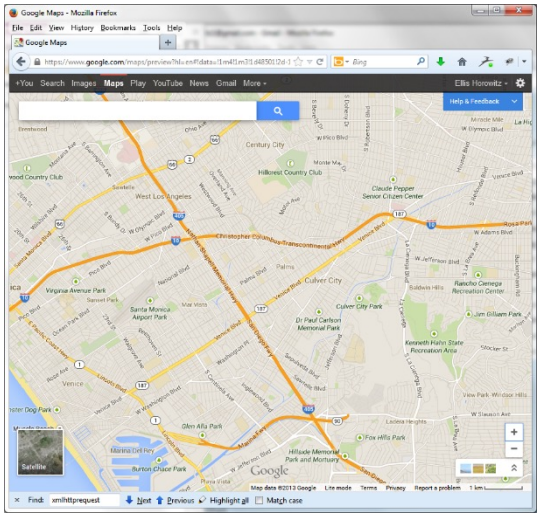
- The **XMLHttpRequest object (XHR)** is the main element of Ajax programming.
- Microsoft first implemented the **XMLHttpRequest** object in **Internet Explorer 5 (IE5)** for Windows as an ActiveX object in **March 1999**, making it the first Ajax-enabled browser.
- Similar functionality is covered in a recommended W3C standard, Document Object Model **(DOM) Level 3 Load and Save** Specification (April 2004):

  http://www.w3.org/TR/DOM-Level-3-LS
- Engineers on the Mozilla project implemented a compatible native version for Mozilla 1.0 (included in Netscape 7, Firefox 1.0 and later releases). Apple has done the same starting with Safari 1.2.
- Other browsers supporting XMLHttpRequest include:
  - Opera 7.6+, Apple Safari 1.2+, all mobile browsers
- XMLHttpRequest moved to W3C in 2006 and back to WHATWG in 2012 as **XMLHttpRequest Living Standard**:
  - https://xhr.spec.whatwg.org/

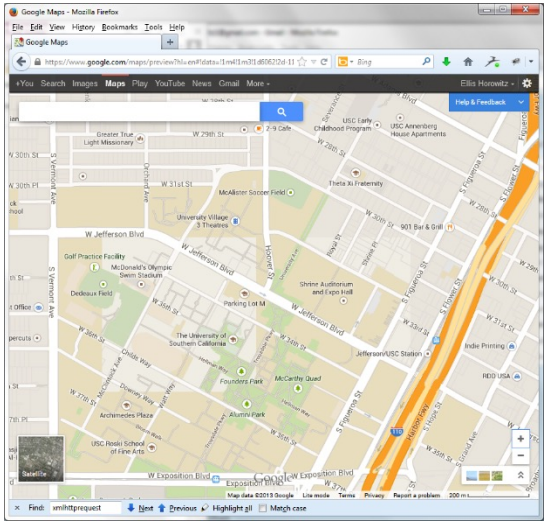
An Example Using Ajax - Google Maps



Initial screen



zoom 3 times



drag map and zoom

See: https://maps.google.com
Notice that the page is never explicitly refreshed. View source and search for XMLHttpRequest; you will find multiple occurrences. (found 2 times on maps.google.com)

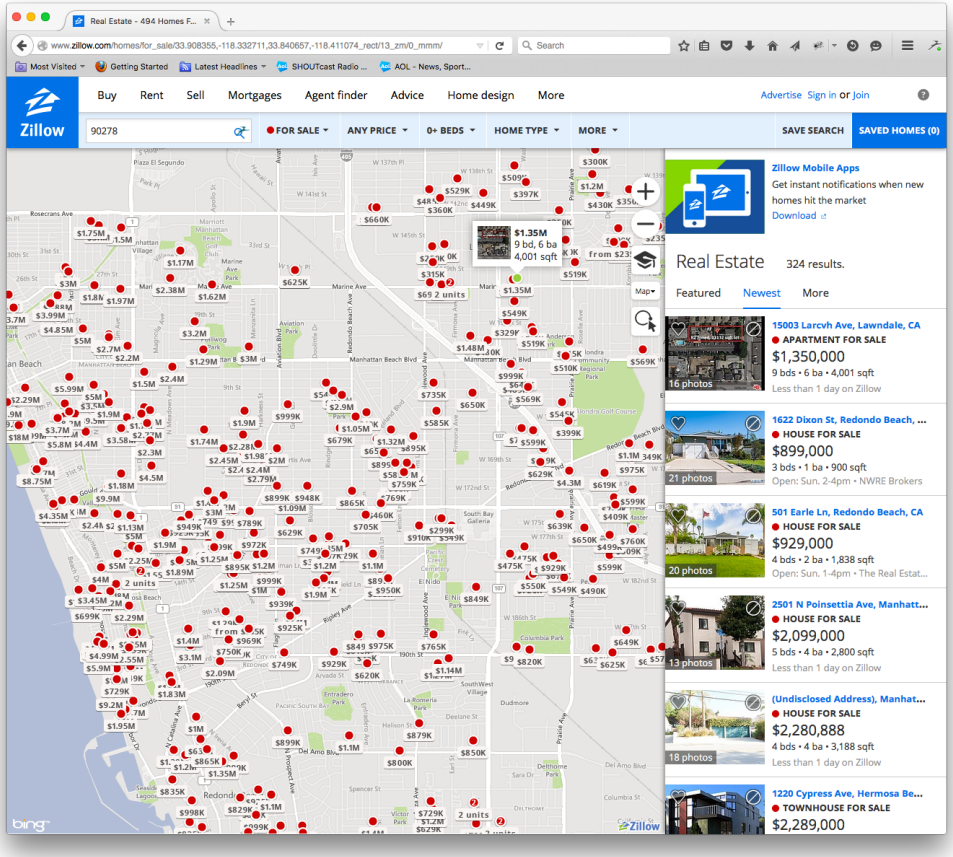
A Mash-Up Combines Multiple Sources of Data

A “mash-up” is a web application that consumes ("remixes") content from different sources and aggregates them to create a new application

Mashup Example – www.zillow.com

A combination of satellite photos with records of home sale prices placed on top of the appropriate houses

Found 4+ references of XMLHttpRequest



Characteristics of Ajax Applications

- They are applications (or Apps), not just web sites
- They allow for smooth, continuous interaction
- "Live" content
- Visual Effects
- Animations, dynamic icons
- Single keystrokes can lead to server calls
- New Widgets (selectors, buttons, tabs, lists)
- New Styles of Interaction (drag-and-drop, keyboard shortcuts, double-click)

Comparing Traditional vs. AJAX Websites

Traditional

- Interface construction is mainly the responsibility of the server
- User interaction is via form submissions
- An entire page is required for each interaction (bandwidth)
- Application is unavailable while an interaction is processing (application speed)

Ajax

- Interface is manipulated by client-side JavaScript manipulations of the Document Object Model (DOM)
- User interaction via HTTP requests occur 'behind the scenes'
- Communication can be restricted to data only
- Application is always responsive

How to Recognize an Ajax Application Internally

"View Source" in the browser and search for:

- Javascript code that invokes:
  - **XMLHttpRequest** or
- JavaScript that "loads" other JavaScript code (files with .js extension)
- XML code passed as text strings to a server, such as '<?xml version="1.0"><page>…</page>'
- Javascript <script> sections that embed code between //<![CDATA[and //]]>
- **IFRAMES** / JavaScript code that creates IFRAMEs, such as window.document.createElement("iframe")
- Use browser "developer tools" to find the code
- **jQuery Ajax** functions (as jQuery.ajax())
- **fetch()**

The Classic Web Application Model

- Most user actions in the browser interface trigger an HTTP request back to a web server.
- The server does some processing — retrieving data, crunching numbers, talking to various legacy systems.
- The server then returns an HTML page to the client.

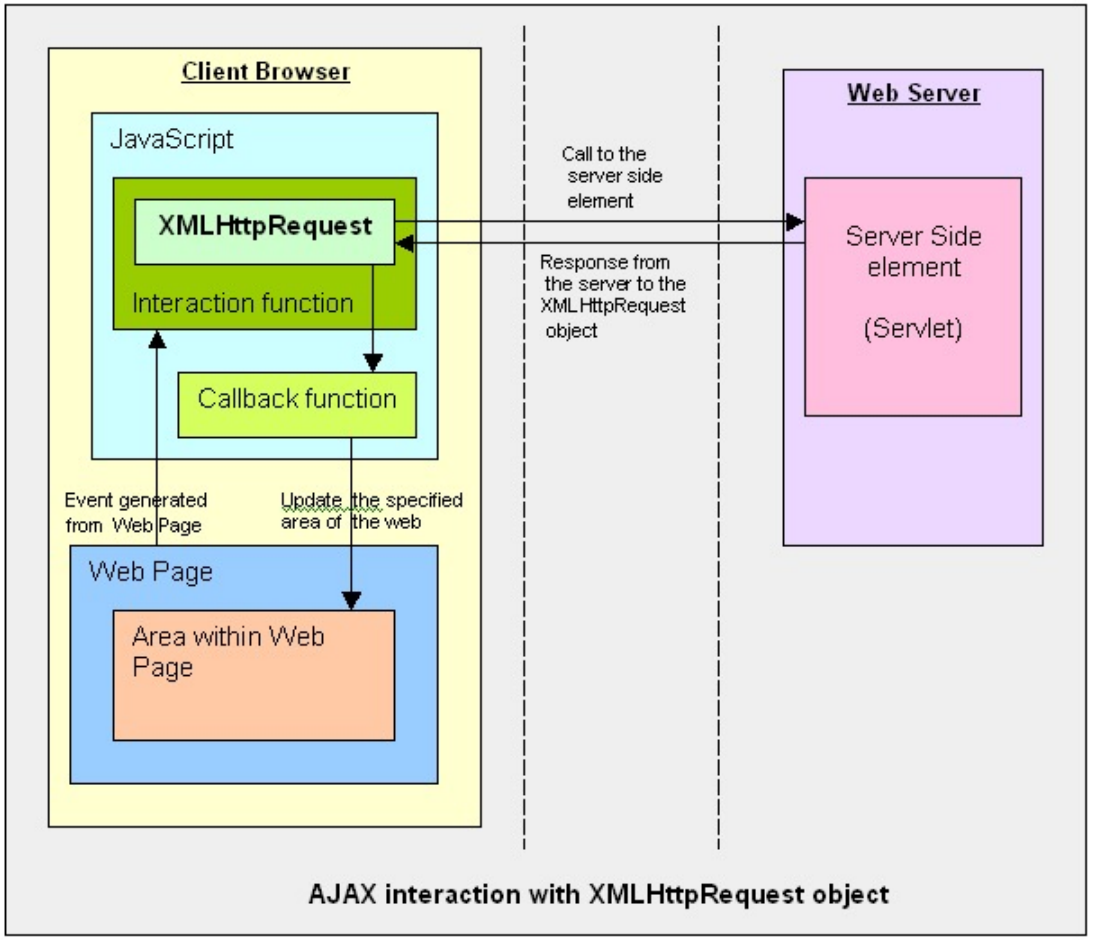
- Approach issues:
  - It doesn't make for a great user experience.
  - While the server is doing its thing, the user is waiting.
  - And at every step in a task, the user waits some more.

The Ajax Web Application Model

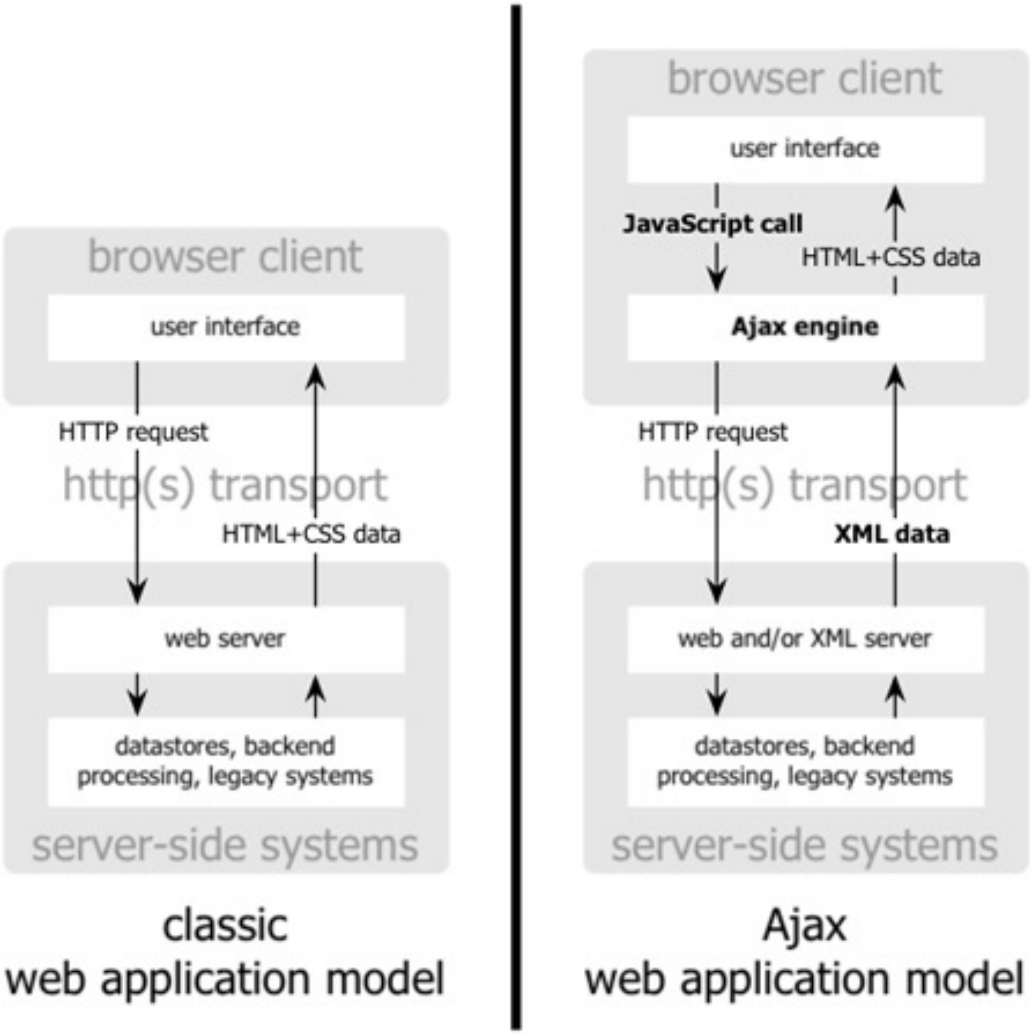
- Ajax introduces an intermediary — an Ajax engine — between the user and the server.
- Instead of loading a webpage, at the start of the session, the browser loads an Ajax engine — written in JavaScript and usually stored in a hidden frame.
- This engine is responsible for
  - rendering the interface that the user sees
  - communicating with the server on the user's behalf.
- The Ajax engine allows the user's interaction with the application to happen asynchronously — independent of communication with the server.
- Approach Benefits:
  - An Ajax application eliminates the start-stop-start-stop nature of interaction on the Web.
  - The user is never staring at a browser window with hourglass, waiting for the server to do something.
  - The application is more responsive.

AJAX:

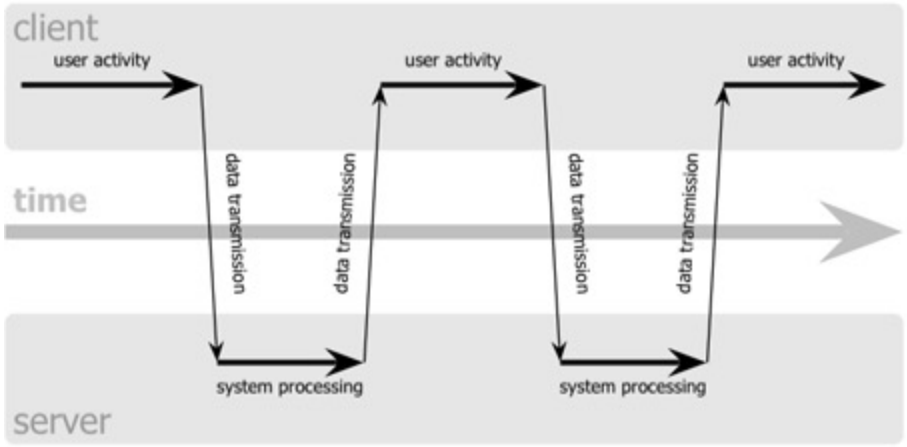
- Cuts down on user wait time
- Uses client to offload some work from the server
- Asynchronous operation



Traditional Web Applications Model compared to the Ajax Model



Classic Web Application Model (synchronous)

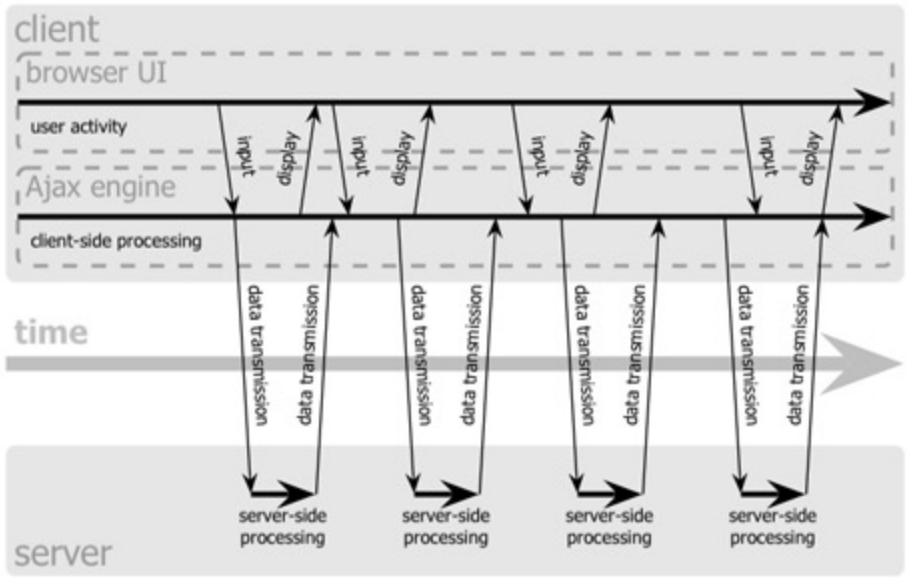


From Jesse James Garrett's "Ajax: a New Approach to Web Applications". See:

http://adaptivepath.org/ideas/ajax-new-approach-web-applications/ (archived below)

https://immagic.com/eLibrary/ARCHIVES/GENERAL/ADTVPATH/A050218G.pdf

Ajax Web Application Model (asynchronous)



Ajax Engine Role

- Every user action that normally would generate an HTTP request takes the form of a JavaScript call to the Ajax engine instead.
- Any response to a user action that doesn't require a trip back to the server, such as:
  - simple data **validation**
  - **editing** data in memory
  - even some **navigation**

  the engine handles on its own.
- If the engine needs something from the server in order to respond, such as:
  - **submitting** data for processing
  - **loading** additional **interface** code
  - **retrieving** new **data**

  the engine makes those requests asynchronously, retrieving results in JSON or XML, without stalling a user's interaction with the application.

Initiating the XMLHttpRequest Object

- Creating an instance of the XMLHttpRequest object requires branching syntax to account for browser differences. For all modern browsers a simple call to the object's constructor function does the job:

```
var req = new XMLHttpRequest();
```

- The object reference returned by both constructors is to an abstract object that works entirely out of view of the user. Its methods control all operations, while its properties hold, among other things, various data pieces returned from the server.

XMLHttpRequest Object Methods

Method	Description
`abort()`	Stops the current request
`getAllResponseHeaders()`	Returns complete set of headers (labels and values) as a string
`getResponseHeader("headerLabel")`	Returns the string value of a single header label
`open("method", "URL"[, asyncFlag[, "userName"[, "password"]]])`	Assigns destination URL, method, and other optional attributes of a pending request
`send(content)`	Transmits the request, optionally with postable string or DOM object data
`setRequestHeader("label", "value")`	Assigns a label/value pair to the header to be sent with a request

See latest requests and responses at: https://xhr.spec.whatwg.org/#request

XMLHttpRequest Object Methods (cont'd)

- Of the methods shown in the Table on the previous slide, the **open()** and **send()** methods are the ones you'll likely use most.
- **open()** sets the scene for an upcoming operation. Two required parameters are the HTTP method you intend for the request and the URL for the connection. For the method parameter, use "GET" on operations that are primarily data retrieval requests; use "POST" on operations that send data to the server, especially if the length of the outgoing data is potentially greater than 512 bytes. The URL may be either a complete or relative URL.
- It is safer to **send** asynchronously and design your code around the onreadystatechange event for the request object. **send** initiates the transaction.

XMLHttpRequest Object Properties

Property	Description
`onreadystatechange`	Event handler for an event that fires at every state change
`readyState`	Object status integer: 0 = uninitialized 1 = loading 2 = loaded 3 = interactive 4 = complete
`responseText`	String version of data returned from server process
`responseXML`	DOM-compatible document object of data returned from server process
`status`	Numeric code returned by server, such as 404 for "Not Found" or 200 for "OK"
`statusText`	String message accompanying the status code

See latest at: https://xhr.spec.whatwg.org/#xmlhttprequest-response

XMLHttpRequest Object Properties (cont'd)

- Use the **readyState** property inside the event handler function that processes request object state change events. While the object may undergo interim state changes during its creation and processing, the value that signals the completion of the transaction is 4.
- Access data returned from the server via the **responseText** or **responseXML** properties. The former provides a string representation of the data, which is used today for **JSON data**. More powerful, however, is the XML document object in the responseXML property. This object is a full-fledged document node object, which can be examined and parsed using W3C DOM node tree methods and properties.
- Note, however, that this is an XML, rather than HTML, document, meaning that you cannot count on the DOM's HTML module methods and properties.
- In today's implementations, everybody is using **responseText**, as this is the way to get **JSON** data.
- The XMLHttpRequest Living Standard includes a **responseType** attribute, that can be set to arraybuffer, blob, document, **json** and text, and a response property that returns a "parsed json object" when json is selected.

XMLHttpRequest Example Code

```
var req;

function loadXMLDoc(url) {
    req = false;
    // branch for native XMLHttpRequest object
    if(window.XMLHttpRequest) {
          try {    req = new XMLHttpRequest();
        } catch(e) {  req = false;
        }
    // branch for IE/Windows ActiveX version (obsolete)
    } else if(window.ActiveXObject) {
          try {
          req = new ActiveXObject("Msxml2.XMLHTTP");
          } catch(e) {
          try {  req = new ActiveXObject("Microsoft.XMLHTTP");
          } catch(e) {   req = false;
          }
          }
    }
    if(req) {
          req.onreadystatechange = processReqChange;
          req.open("GET", url, true);
          req.send("");
    }
}
```

This code instantiates an XmlHttpRequest object depending upon the browser

onreadystatechange Event Handler Function

```
function processReqChange() { // see previous slide
    // only if req shows "loaded"
    if (req.readyState == 4) {
        // only if "OK"
        if (req.status == 200) {
            // processing statements req.responseText
            // for JSON
            // and req.responseXML for XML go here...
        } else {
alert("There was a problem retrieving the data:\n" +
   req.statusText);
        }
    }
}
```

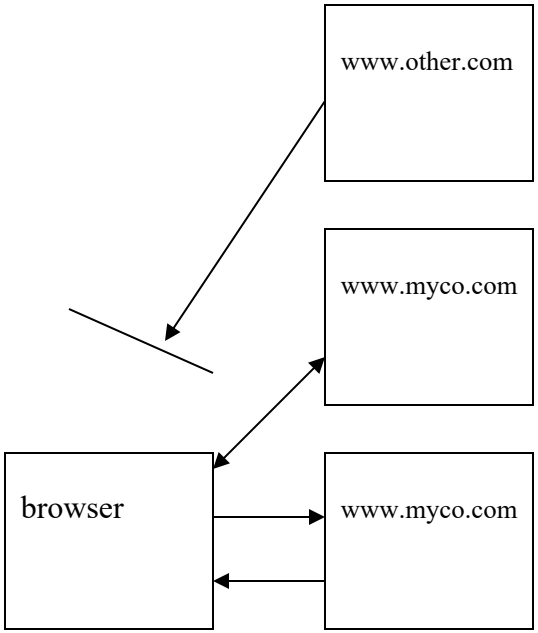
Security Issues

- When the XMLHttpRequest object operates within a browser, it **adopts the same-domain security policies** of typical JavaScript activity (sharing the same "sandbox," as it were).

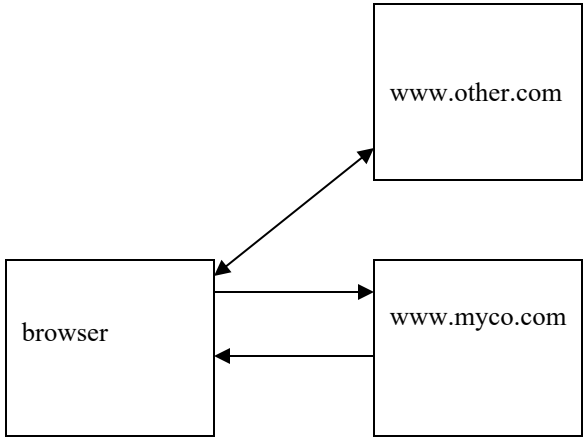
- First, on most browsers supporting this functionality, the page that bears scripts accessing the object needs to be retrieved via **http: protocol**, meaning that **you won't be able to test the pages from a local hard disk** (**file: protocol**) without some extra security issues cropping up, especially in Mozilla and IE on Windows.

- Second, the domain of the URL request destination must be the same as the one that serves up the page containing the script. This means, unfortunately, that client-side scripts cannot fetch web service data from other sources and blend that data into a page. **Everything must come from the same domain.**

AJAX Cross Domain Security



For security reasons, scripts are only allowed to access data which comes from the same domain



The one exception is for images: images can come from any domain, without any security risk.

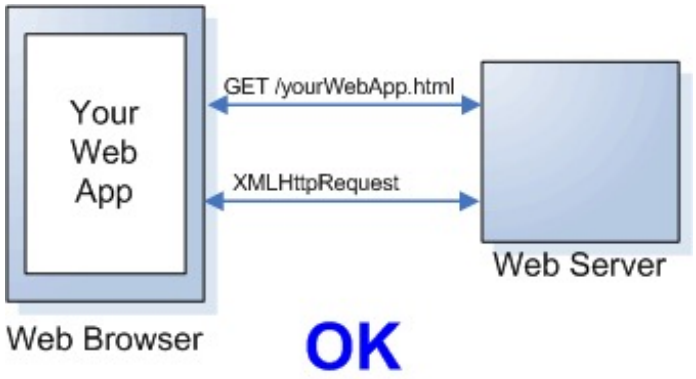
This is why all the mash-up applications involve images

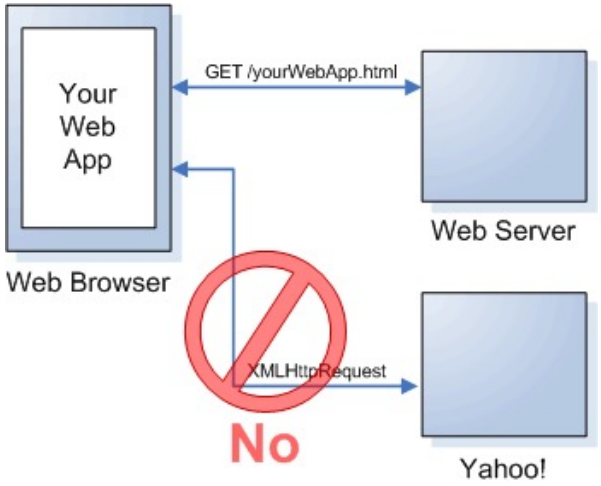
They simply would not be possible for other kinds of data

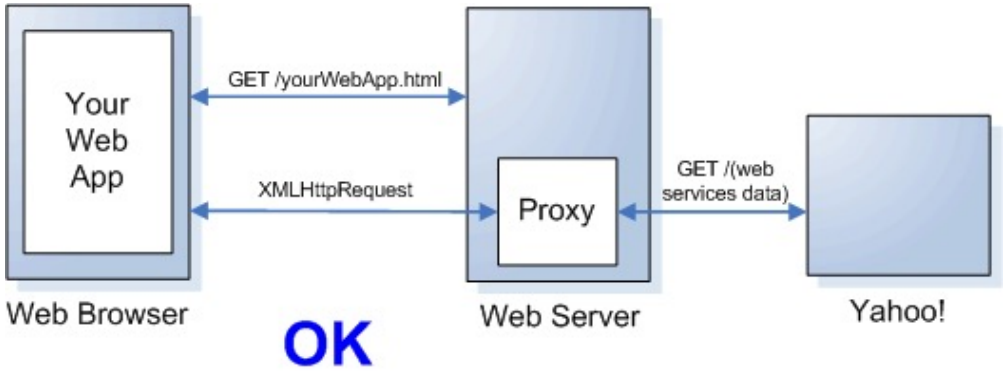
Cross-domain Restrictions and a Solution

- Browser security restrictions prevent your web application from opening network connections to domains other than the one your application came from.
- For example, suppose your web application wants to use data both from your site and from Yahoo!; normally this is not possible as it is a violation of browser cross-domain security policy.
- **One way** to work around this issue is to install a **web proxy** on your server that will pass requests from your application to Yahoo! and the data back again. This is what is used in our assignments.
- If you are using a proxy to relay requests from your web application to Yahoo!, the actual request URL you use from your web application is different, as you must relay your request through your web server proxy.
- **Another way** is **CORS**, which works on all recent browsers. Of course, you will need a server that you trust and that is set up to accept CORS requests. (see slides later in this set)

Why You Need a Proxy







Alternative: Fetch API

- The **Fetch API** provides a JavaScript interface for accessing and manipulating requests and responses. It also provides a global **fetch()** method to fetch resources **asynchronously**.
- Fetch also provides a single logical place to define other HTTP-related concepts such as **CORS** and **HTTP extensions**.
- The fetch specification differs from **jQuery.ajax()** in three main ways:
  - The Promise returned from fetch() won't reject on HTTP error status even if the response is an HTTP 404 or 500.
  - fetch() won't receive cross-site cookies; you can't establish a cross site session using fetch. Set-Cookie headers from other sites are silently ignored.
  - Fetch() won't send cookies, unless you set the credentials init option. won't receive cross-site cookies;

```
1  fetch('http://example.com/movies.json')
2    .then((response) => {
3      return response.json();
4    })
5    .then((data) => {
6      console.log(data);
7    });
```

Alternative: Fetch API (cont'd)

- see: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
- The fetch() method can optionally accept a second parameter, an init object that allows you to control a few settings:

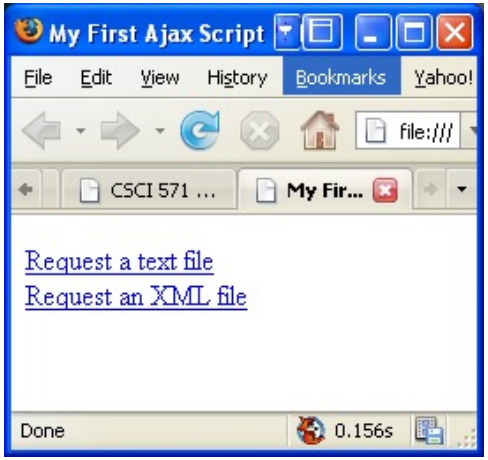
```
// Example POST method implementation:
async function postData(url = '', data = {}) {
  // Default options are marked with *
  const response = await fetch(url, {
    method: 'POST', // *GET, POST, PUT, DELETE, etc.
    mode: 'cors', // no-cors, *cors, same-origin
    cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached
    credentials: 'same-origin', // include, *same-origin, omit
    headers: {
      'Content-Type': 'application/json'
      // 'Content-Type': 'application/x-www-form-urlencoded',
    },
    redirect: 'follow', // manual, *follow, error
    referrerPolicy: 'no-referrer', // no-referrer, *client
    body: JSON.stringify(data) // body data type must match "Content-Type" header
  });
  return await response.json(); // parses JSON response into native JavaScript objects
}

postData('https://example.com/answer', { answer: 42 })
  .then((data) => {
    console.log(data); // JSON data parsed by `response.json()` call
  });
```

A First Ajax Example – Using Ajax to Download Files

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
   Transitional//EN">
<html><head>
   <title>First Ajax Script</title>
   <script src="script01.js" type="text/javascript"
   language="Javascript">
   </script>
</head><body>
   <p><a id="makeTextRequest" href="gAddress.txt">Request a
   text file</a><br />
   <a id="makeXMLRequest" href="us-states.xml">
Request an XML file</a></p>
   <div id="updateArea"> </div>
</body>
</html>
```

The javascript file does all of the work



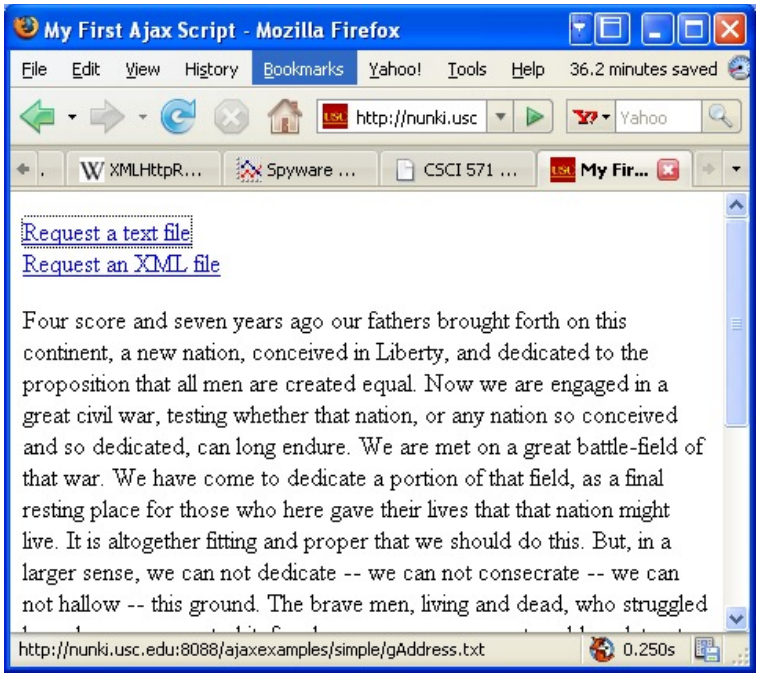
Imported JavaScript-script01.js

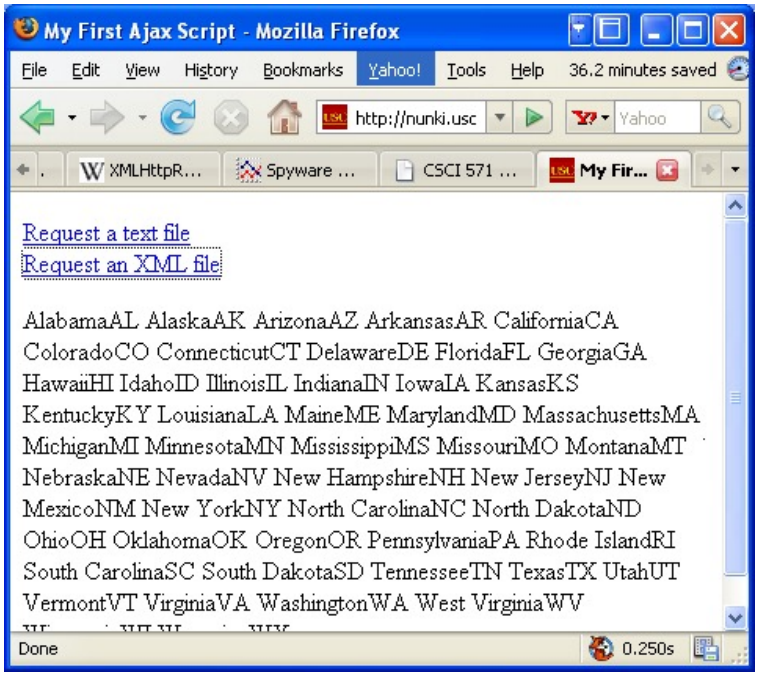
```
window.onload = initAll;
var xhr = false;
function initAll() {
    document.getElementById("makeTextRequest").onclick = getNewFile;
    document.getElementById("makeXMLRequest").onclick = getNewFile;}
function getNewFile() {
    makeRequest(this.href);  return false;}
function makeRequest(url) {
    if (window.XMLHttpRequest) {  xhr = new XMLHttpRequest();}
    else { if (window.ActiveXObject) {
                    try { xhr = new ActiveXObject("Microsoft.XMLHTTP"); }
                    catch (e) { }
          }   }
    if (xhr) {  xhr.onreadystatechange = showContents;
          xhr.open("GET", url, true);  xhr.send(null);  }
    else { document.getElementById("updateArea").innerHTML = "Sorry, but I couldn't
    create an XMLHttpRequest";  }  }
function showContents() {
    if (xhr.readyState == 4) {
          if (xhr.status == 200) {
                    var outMsg = (xhr.responseXML &&
    xhr.responseXML.contentType=="text/xml") ?
    xhr.responseXML.getElementsByTagName("choices")[0].textContent : xhr.responseText;
    } else { var outMsg = "There was a problem with the request " + xhr.status; }
          document.getElementById("updateArea").innerHTML = outMsg;   } }
```

On page load the onclick event is set to call the function When the click is made, getNewFile and makerequest are executed.

showContents waits for a successful return of an file; it then prints the result in the browser

Browser Output





Result of clicking on the first link

Result of clicking on the second link

http://csci571.com/ajaxexamples/simple/script01.html

Second Ajax Example – Using Ajax to Download Files from Flickr

- **Here is the html file, which basically loads script02.js**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN">
<html><head><title>Second Ajax Script</title>
<script src="script02.js" type="text/javascript" language="Javascript"></script>
</head><body><div id="pictureBar"> </div></body></html>
```

- **Here is script02.js**

```
window.onload = initAll;
var xhr = false;
function initAll() {
    if (window.XMLHttpRequest) { xhr = new XMLHttpRequest(); }
    else { if (window.ActiveXObject) {
          try { xhr = new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) { }  } }
    if (xhr) { xhr.onreadystatechange = showPictures;
          xhr.open("GET", "flickrfeed.xml", true);   xhr.send(null);  }
    else {  alert("Sorry, but I couldn't create an XMLHttpRequest");  }  }
function showPictures() {
    var tempDiv = document.createElement("div");
    var pageDiv = document.getElementById("pictureBar");
    if (xhr.readyState == 4) {
          if (xhr.status == 200) {
                tempDiv.innerHTML = xhr.responseText;
                var allLinks = tempDiv.getElementsByTagName("a");
                      for (var i=1; i<allLinks.length; i+=2) {
                pageDiv.appendChild(allLinks[i].cloneNode(true)); }  }
    else { alert("There was a problem with the request " + xhr.status); }  }  }
```

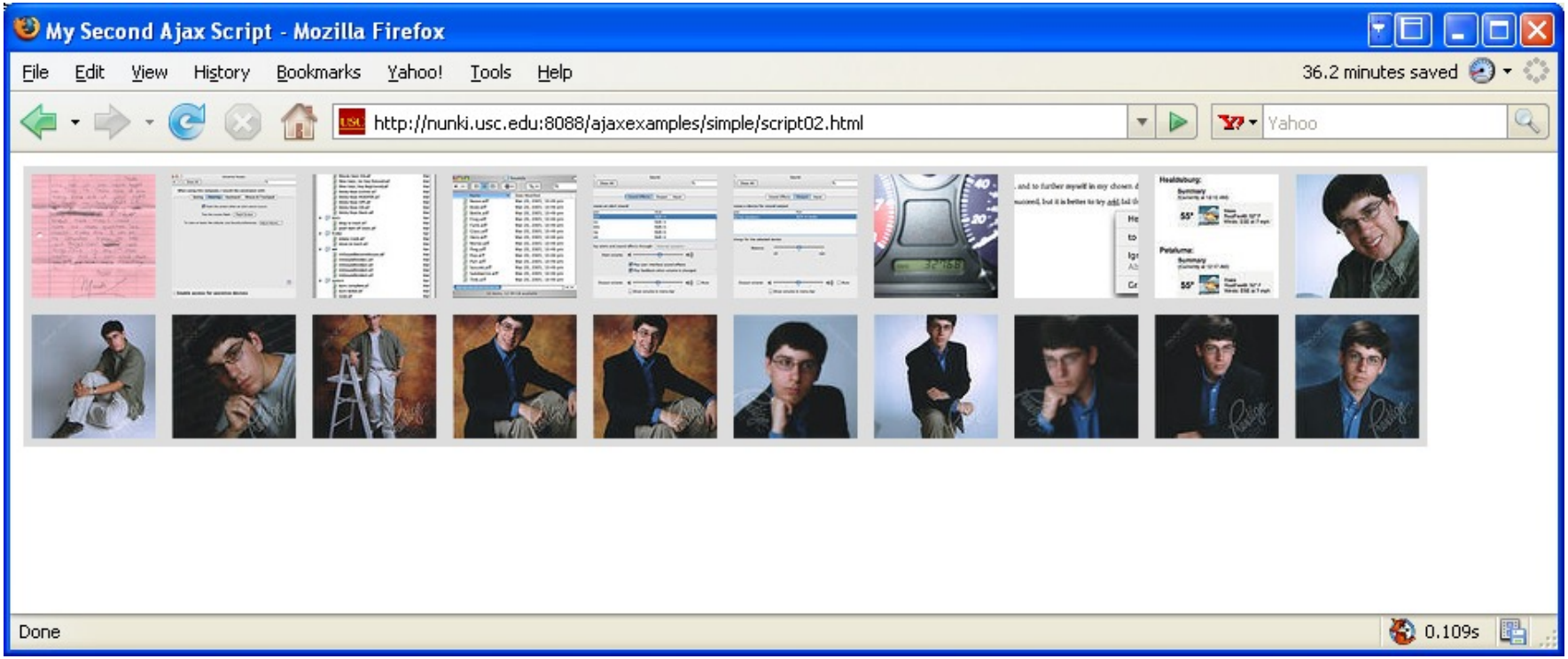
ShowPictures retrieves an file from flickr;
The result is extracted from responseText and assigned to innerHTML property

Portion of Flickr XML file

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<feed xmlns=http://www.w3.org/2005/Atom
Xmlns=http://purl.org/dc/elements/1.1>
<title>Dori Simth's Photos</title>
<link rel="self"
    href=http://www.flickr.com/services/feeds/photos_public.gne?id=23922109@N00 />
<link rel="alternate" type="text/html" href=http://www.flickr.com/photos/dorismith/
    />
<id>tag:flickr.com,2005:/photos/public/116078</id>
<icon>http://static.flickr.com/5/buddyicons/23922109@N00.jpg?1113973282</icon>
<subtitle>A feed of Dori Smith's Photos</subtitle>
<updated>2006-03-22T20:12:44Z</updated>
<generator uri=http://www.flickr.com/>Flickr</generator>
<entry>
<title>Mash note</title>
<link rel="alternate" type="text/html"
    href=http://www.flickr.com/photos/dorismith/116463569/ />
OTHER STUFF
<p> <a href=http://www.flickr.com/photos/dorismith/116463569/ title="Mash
    note"><img src=http://static.flickr.com/44/116463569_483fd4ee7c_s.jpg
    width="75" height="75" alt="Mash note" style="border: 5px solid #ddd;" /></a>
    </p>
```

Each <entry> node has two links; This application uses the second link so the showPictures loop starts with 1 rather than 0 and increments by 2; each link contains the thumbnail image inside it; every thumbnail is a link back to the original photo

Browser Output



http://csci571.com/ajaxexamples/simple/script02.html

Third Ajax Example - Refreshing Server Data

- This extension retrieves a new version of the data from the server, refreshing the page; **here is the html accessing javascript**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN">
<html><head><title>My Third Ajax Script</title>
<script src="script03.js" type="text/javascript" language="Javascript"></script></head>
<body><div id="pictureBar"> </div></body></html>
```

- **And here is the source for script03.js**

```
window.onload = initAll;
var xhr = false;
function initAll() {  same as previously except it calls getPix }
function getPix() {  xhr.open("GET", "flickrfeed.xml", true);
xhr.onreadystatechange = showPictures;  xhr.send(null);setTimeout("getPix()",5 * 1000); }
function showPictures() {
    var tempDiv = document.createElement("div");
    var tempDiv2 = document.createElement("div");
    if (xhr.readyState == 4) {
          if (xhr.status == 200) {
          tempDiv.innerHTML = xhr.responseText;
          var allLinks = tempDiv.getElementsByTagName("a");
          for (var i=1; i<allLinks.length; i+=2) {
                    tempDiv2.appendChild(allLinks[i].cloneNode(true)); }
          allLinks = tempDiv2.getElementsByTagName("a");
          var randomImg = Math.floor(Math.random() * allLinks.length);
    document.getElementById("pictureBar").innerHTML = allLinks[randomImg].innerHTML;
          } else { alert("There was a problem with the request " + xhr.status); } }  }
```

The call to getPix is placed in setTimeout which causes repeated execution, every 5 seconds;
An array of links of photographs is created, a random number computed, and use it as an index into the array

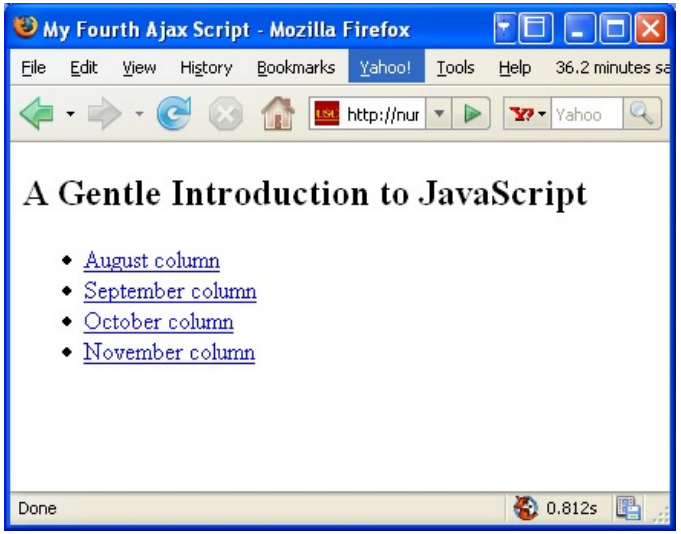
Browser Output

Three consecutive outputs

http://csci571.com/ajaxexamples/simple/script03.html

Fourth Ajax Example - Previewing Links

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
    Transitional//EN">
<html><head>
    <title>My Fourth Ajax Script</title>
    <link rel="stylesheet" rev="stylesheet"
    href="script04.css" />
    <script src="script04.js"
    type="text/javascript" language="Javascript">
    </script>
</head><body>
<h2>A Gentle Introduction to JavaScript</h2><ul>
    <li><a href="jsintro/2000-08.html">August
    column</a></li>
    <li><a href="jsintro/2000-09.html">September
    column</a></li>
    <li><a href="jsintro/2000-10.html">October
    column</a></li>
    <li><a href="jsintro/2000-11.html">November
    column</a></li>
</ul>
<div id="previewWin"> </div>
</body>
</html>
```



http://csci571.com/ajaxexamples/simple/script04.html

The stylesheet

```
#previewWin {
   background-color: #FF9;
   width: 400px;
   height: 100px;
   font: .8em arial, helvetica, sans-serif;
   padding: 5px;
   position: absolute;
   visibility: hidden;
   top: 10px;
   left: 10px;
   border: 1px #CC0 solid;
   clip: auto;
   overflow: hidden;
}

#previewWin h1, #previewWin h2 {
   font-size: 1.0em;
}
```

The javascript source

```
window.onload = initAll;
var xhr = false;
var xPos, yPos;
function initAll() {
    var allLinks = document.getElementsByTagName("a");
    for (var i=0; i< allLinks.length; i++) {
          allLinks[i].onmouseover = showPreview; } }
function showPreview(evt) { getPreview(evt); return false; }
function hidePreview() {
      document.getElementById("previewWin").style.visibility = "hidden"; }
function getPreview(evt) {
    if (evt) { var url = evt.target; }
    else { evt = window.event;  var url = evt.srcElement; }
    xPos = evt.clientX;  yPos = evt.clientY;
    if (window.XMLHttpRequest) {
          xhr = new XMLHttpRequest(); }
    else { if (window.ActiveXObject) {
          try {  xhr = new ActiveXObject("Microsoft.XMLHTTP"); }  catch (e) { }  } }
    if (xhr) { xhr.onreadystatechange = showContents;
          xhr.open("GET", url, true);  xhr.send(null);
    } else { alert("Sorry, but I couldn't create an XMLHttpRequest"); }  }
```

The javascript source cont'd

```
function showContents() {
    var prevWin = document.getElementById("previewWin");
    if (xhr.readyState == 4) {
          prevWin.innerHTML = (xhr.status == 200) ? xhr.responseText : "There was
    a problem with the request " + xhr.status;
          prevWin.style.top = parseInt(yPos)+2 + "px";
          prevWin.style.left = parseInt(xPos)+2 + "px";
          prevWin.style.visibility = "visible";
          prevWin.onmouseout = hidePreview; }}
```

Notes: initall goes through all of the links and adds an onmouseover event;
showPreview( ) and hidePreview( ) are both needed; the latter sets the preview window back to hidden;
In getPreview( ), depending upon the browser, the URL is in either evt.target or in window.event.srcElement; the (x,y) position is extracted;
In showContents( ) the data is placed in prevWin.innerHTML from responseText;
The preview window is placed just below and to the right of the cursor position that triggered the call

Fifth Ajax Example, Auto Completion

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN">
<html><head><title>My Fifth Ajax Script</title>
   <link rel="stylesheet" rev="stylesheet" href="script05.css" />
   <script src="script05.js" type="text/javascript"
   language="Javascript">
   </script>
</head><body>
   <form action="#">
   Please enter your state:<br />
   <input type="text" id="searchField" autocomplete="off" /><br />
        <div id="popups"> </div>
   </form></body></html>
```

Autocomplete attribute is set to off to prevent browsers from trying to autocomplete the field

Initial screen

The stylesheet

```
body, #searchfield {
    font: 1.2em arial, helvetica, sans-serif;
}
.suggestions {
    background-color: #FFF;
    padding: 2px 6px;
    border: 1px solid #000;
}
.suggestions:hover {
    background-color: #69F;
}
#popups {
    position: absolute;
}
#searchField.error {
    background-color: #FFC;
}
```

The JavaScript Source

Onkeyup captures single keystrokes

```
window.onload = initAll;
var xhr = false;   var statesArray = new Array();
function initAll() {
    document.getElementById("searchField").onkeyup = searchSuggest;
    if (window.XMLHttpRequest) {  xhr = new XMLHttpRequest(); }
    else { if (window.ActiveXObject) {
          try { xhr = new ActiveXObject("Microsoft.XMLHTTP"); } catch (e) { } }}
    if (xhr) {
          xhr.onreadystatechange = setStatesArray;
          xhr.open("GET", "us-states.xml", true); xhr.send(null);
    } else { alert("Sorry, but I couldn't create an XMLHttpRequest"); }}
function setStatesArray() {
    if (xhr.readyState == 4) {
          if (xhr.status == 200) {
                    if (xhr.responseXML) {
          var allStates = xhr.responseXML.getElementsByTagName("item");
                              for (var i=0; i<allStates.length; i++) {
                                        statesArray[i] =
    allStates[i].getElementsByTagName("label")[0].firstChild; } }  }
    else { alert("There was a problem with the request " + xhr.status); } }  }
```

The example uses the xml file listing the states

Here we read the list of states and place them in an array

The JavaScript Source cont'd

```
function searchSuggest() {
    var str = document.getElementById("searchField").value;
    document.getElementById("searchField").className = "";
    if (str != "") {
          document.getElementById("popups").innerHTML = "";
          for (var i=0; i<statesArray.length; i++) {
                    var thisState = statesArray[i].nodeValue;
                              if
    (thisState.toLowerCase().indexOf(str.toLowerCase()) == 0) {
          var tempDiv = document.createElement("div");
          tempDiv.innerHTML = thisState;
          tempDiv.onclick = makeChoice;
          tempDiv.className = "suggestions";
          document.getElementById("popups").appendChild(tempDiv); }   }
          var foundCt = document.getElementById("popups").childNodes.length;
          if (foundCt == 0) {
                    document.getElementById("searchField").className = "error"; }
          if (foundCt == 1) {
                    document.getElementById("searchField").value =
    document.getElementById("popups").firstChild.innerHTML;
                    document.getElementById("popups").innerHTML = ""; }   }   }
function makeChoice(evt) {
    var thisDiv = (evt) ? evt.target : window.event.srcElement;
    document.getElementById("searchField").value = thisDiv.innerHTML;
    document.getElementById("popups").innerHTML = ""; }
```

This routine is called on a key up; The value in the search field is first extracted; if nothing is entered, do nothing;

If indexof returns 0, then we have a hit;

Add a state to the list of possibilities

Foundct is the number of matches

Unique hit, place it in proper place

Browser Output

Initial screen

3 examples

http://csci571.com/ajaxexamples/simple/script05.html

Some References

- Ajax (programming) – Wikipedia: http://en.wikipedia.org/wiki/AJAX
- Using the XML HTTP Request object: http://jibbering.com/2002/4/httprequest.html
- XMLHttpRequest & Ajax Working Examples: http://www.fiftyfoureleven.com/resources/programming/xmlhttprequest/examples
- Very Dynamic Web Interfaces: http://www.xml.com/pub/a/2005/02/09/xml-http-request.html

Ajax Enabled Technologies (Toolkits)

- **Ruby** on Rails:
  http://www.rubyonrails.org/
- Microsoft **ASP.NET** Ajax:
  https://docs.microsoft.com/en-us/aspnet/ajax/
- DevExpress **AJAX Control Toolkit** for ASP.NET Forms:
  https://github.com/DevExpress/AjaxControlToolkit
- **JQuery**:
  http://jquery.com
- Google -- **Angular**:
  http://angular.io (2.x-10.x)

Browser Security Features
(jump ahead to CORS slides)
(skip optional slides 48-55)

Credits

- The following material is based on the google wiki, Browser Security Handbook:

  https://code.google.com/p/browsersec/wiki/Part1

  https://code.google.com/p/browsersec/wiki/Part2

Part1 Outline

- Basic concepts behind web browsers
  - Uniform Resource Locators
    - Unicode in URLs
  - True URL schemes
  - Pseudo URL schemes
  - Hypertext Transfer Protocol
  - Hypertext Markup Language
    - HTML entity encoding
  - Document Object Model
  - Browser-side Javascript
    - Javascript character encoding
  - Other document scripting languages
  - Cascading stylesheets
    - CSS character encoding
  - Other built-in document formats
  - Plugin-supported content

Part2 Outline

- Standard browser security features
  - Same-origin policy
    - Same-origin policy for DOM access
    - Same-origin policy for XMLHttpRequest
    - Same-origin policy for cookies
    - Same-origin policy for Flash
    - Same-origin policy for Java
    - Same-origin policy for Silverlight
    - Same-origin policy for Gears
    - Origin inheritance rules
    - Cross-site scripting and same-origin policies
  - Life outside same-origin rules
    - Navigation and content inclusion across domains
    - Arbitrary page mashups (UI redressing)
    - Gaps in DOM access control
    - Privacy-related side channels
  - Various network-related restrictions
    - Local network / remote network divide
    - Port access restrictions
    - URL scheme access rules
    - Etc

Same-origin policy for DOM access

- the term "same-origin policy" most commonly refers to a mechanism that governs the ability for Javascript and other scripting languages to access DOM properties and methods across domains
- the same-origin model attempts to ensure proper separation between unrelated pages, and serve as a method for sandboxing potentially untrusted or risky content within a particular domain

Three-Step Decision Process

- the model boils down to this three-step decision process:

1. If protocol, host name, and - for browsers other than Microsoft Internet Explorer - port number for two interacting pages match, access is granted with no further checks.
2. Any page may set the ***document.domain*** parameter to a right-hand, fully-qualified fragment of its current host name (e.g., ***foo.bar.example.com*** may set it to ***example.com***, but not ***ample.com***). If two pages explicitly and *mutually* set their respective ***document.domain*** parameters to the same value, and the remaining same-origin checks are satisfied, access is granted.
3. If neither of the above conditions is satisfied, access is denied.

Drawbacks of Same-Origin Policy

- once any two legitimate subdomains in ***example.com***, e.g. ***www.example.com*** and ***payments.example.com***, choose to cooperate, any other resource in that domain, such as ***user-pages.example.com***, may then set its own ***document.domain*** likewise, and arbitrarily mess with ***payments.example.com***. This means that in many scenarios, ***document.domain*** may not be used safely at all.
- Whenever ***document.domain*** cannot be used - either because pages live in completely different domains, or because of the above problem - legitimate client-side communication between, for example, embeddable page gadgets, is completely forbidden in theory, and in practice very difficult to arrange
- Whenever tight integration of services within a single host name is pursued to overcome these communication problems, because of the inflexibility of same-origin checks, there is no usable method to sandbox any untrusted or particularly vulnerable content to minimize the impact of security problems.

Special Cases that Are *Omitted* From the Policy

- The ***document.domain*** behavior when hosts are addressed by IP addresses, as opposed to fully-qualified domain names, is not specified.
- The ***document.domain*** behavior with extremely vague specifications (e.g., ***co.uk***) is not specified.
- The algorithms of context inheritance for pseudo-protocol windows, such as ***about:blank***, are not specified.
- The behavior for URLs that do not meaningfully have a host name associated with them (e.g., ***file://***) is not defined, causing **some browsers** to permit locally saved files to access every document on the disk or on the web; users are generally not aware of this risk, potentially exposing themselves.
- The behavior when a single name resolves to vastly different IP addresses (for example, one on an internal network, and another on the Internet) is not specified, permitting various attacks and tricks

Same-origin policy for XMLHttpRequest

- security-relevant features provided by ***XMLHttpRequest***
  - The ability to specify an arbitrary HTTP request method (via the ***open()*** method),
  - The ability to set custom HTTP headers on a request (via ***setRequestHeader()***),
  - The ability to read back full response headers (via ***getResponseHeader()*** and ***getAllResponseHeaders()***),
  - The ability to read back full response body as Javascript string (via ***responseText*** property).

Checks on XMLHttpRequest

- The set of checks implemented in all browsers for ***XMLHttpRequest*** is a close variation of DOM same-origin policy, with the following changes:
- Checks for ***XMLHttpRequest*** targets do not take ***document.domain*** into account, making it impossible for third-party sites to mutually agree to permit cross-domain requests between them.
- In some implementations, there are additional restrictions on protocols, header fields, and HTTP methods for which the functionality is available, or HTTP response codes which would be shown to scripts (see later).

Cross-origin resource sharing (CORS)

Cross-origin resource sharing (CORS) allows many resources (e.g, fonts, JavaScript, etc.) on a web page to be requested across domains. In particular, AJAX calls can use XMLHttpRequest across domains.

The CORS standard adds new HTTP headers. If the browser recognizes a cross-domain request, it sends an "Origin" HTTP header. Suppose a page from http://www.social-network.com attempts to access user data from online-personal-calendar.com. If the browser supports CORS, this header is sent:

Origin: http://www.social-network.com

If the server at online-personal-calendar.com allows the request, it sends an Access-Control-Allow-Origin (ACAO) header in the response. The value of the header indicates what origin sites are allowed. For example:

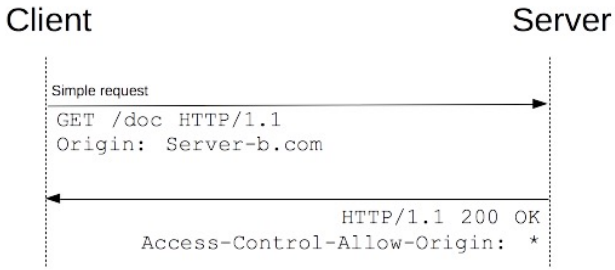
Access-Control-Allow-Origin: http://www.social-network.com

Access-Control-Allow-Origin: *

If the server does not allow the CORS request, the browser will deliver an error instead of the online-personal-calendar.com response. **Firefox 3.5+, Safari 4+, Chrome3+, IE 10+, Opera 12+, and Edge support CORS**. See:

https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS
http://enable-cors.org/server_apache.html

CORS Example




```
1  GET /resources/public-data/ HTTP/1.1
2  Host: bar.other
3  User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5  Accept-Language: en-us,en;q=0.5
6  Accept-Encoding: gzip,deflate
7  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
8  Connection: keep-alive
9  Referer: http://foo.example/examples/access-control/simpleXSInvocation.html
10 Origin: http://foo.example
11
12
13 HTTP/1.1 200 OK
14 Date: Mon, 01 Dec 2008 00:23:53 GMT
15 Server: Apache/2.0.61
16 Access-Control-Allow-Origin: *
17 Keep-Alive: timeout=2, max=100
18 Connection: Keep-Alive
19 Transfer-Encoding: chunked
20 Content-Type: application/xml
21
22 [XML Data]
```