+

JSON, AJAX

+

JSON: The Basics

JSON is…

- "JSON" stands for "JavaScript Object Notation"
- A lightweight text based data-interchange format
- Completely language independent
- Based on a subset of the JavaScript Programming Language
- Easy to understand, manipulate and generate

JSON is NOT…

- Overly Complex
- A “document” format
- A markup language
- A programming language

Why use JSON?

- Straightforward syntax, language independent
  - Plain text formats
  - “Self-describing“ (human readable)
  - Less syntax, no semantics
  - Hierarchical (Values can contain lists of objects or values)
  - JSON includes arrays
  - Names in JSON must not be JavaScript reserved words
  - JSON uses typed objects
  - Lack of namespaces

Why use JSON? (…)

- Easy to create and manipulate
  - Can be natively parsed in JavaScript using **eval()**
  - Supported by all major JavaScript frameworks
  - Supported by most backend technologies
  - Can be used by Ajax
  - Properties are immediately accessible to JavaScript code
  - No inherit validation (but there is JSONlint)

JSON example

```
{"students":[
  {"name":"John", "age":"23", "city":"Agra"},
  {"name":"Steve", "age":"28", "city":"Delhi"},
  {"name":"Peter", "age":"32", "city":"Chennai"},
  {"name":"Chaitanya", "age":"28", "city":"Bangalore"}
]}
```

JSON Object Syntax

- Unordered sets of name/value pairs
  - Begins with **{** (left brace)
  - Ends with **}** (right brace)
  - Each name is followed by **:** (colon)
  - Name/value pairs are separated by **,** (comma)

```
var employeeData = {
  "employee_id": 1234567,
  "name": "Jeff Fox",
  "hire_date": "1/1/2013",
  "location": "Norwalk, CT",
  "consultant": false
};
```

Arrays in JSON

- An ordered collection of values
  - Begins with **[** (left bracket)
  - Ends with **]** (right bracket)
  - Name/value pairs are separated by **,** (comma)

```
var employeeData = {
  "employee_id": 1236937,
  "name": "Jeff Fox",
  "hire_date": "1/1/2013",
  "location": "Norwalk, CT",
  "consultant": false,
   "random_nums": [ 24,65,12,94 ]
};
```

Data Types

- Strings
  - Sequence of 0 or more Unicode characters
  - Wrapped in "double quotes“
- Numbers
  - Integer
  - Real
  - Scientific
  - No octal or hex
  - No NaN or Infinity – Use **null** instead.
- Booleans: true or false
- Null: A value that specifies nothing or no value.

How & When to use JSON

- Transfer data to and from a server
- Perform asynchronous data calls without requiring a page refresh
- Working with data stores
- Compile and save form or user data for local storage

+
The simplest JSON example

```
<!DOCTYPE html>
<html>
<body>
<p id="user"></p>

<script>
var s = '{"first_name" : "Sammy", "last_name" : "Shark", "location" : "Ocean"}';

var obj = JSON.parse(s);

document.getElementById("user").innerHTML =
"Name: " + obj.first_name + " " + obj.last_name + "<br>" +
"Location: " + obj.location;
</script>

</body>
</html>
```

Serializing

- Turning an object into a string is known as *serializing an object*.
  - It is incredibly useful because it allows you to store objects in plain text files, or transfer them between web applications, even if those applications are written in different languages.
  - Once the object is serialized, you can give it to the other program, and if the other program knows that the format is JSON, the program can *deserialize* the object (turn the string back into an object again)

+ stringify()

```
function Pet(type, name, weight, likes) {
            this.type = type;
            this.name = name;
            this.weight = weight;
            this.likes = likes;
            }

window.onload = init;

function init() {
            var pickles = new Pet("cat", "Pickles", 7, ["sleeping", "purring", "eating butter"] );
            console.log(pickles);
            var picklesJSON = JSON.stringify(pickles);
            console.log(picklesJSON);
            var tilla = new Pet("dog", "Tilla", 25, ["sleeping","eating","walking"]);
            console.log(tilla);
            var tillaJSON = JSON.stringify(tilla);
            console.log(tillaJSON);
}
```

- **stringify()** method of the **JSON** object, and pass in the **pickles** object as the argument.
- We get back a JSON version of the object, which we store in the variable **picklesJSON**.
- We did the same thing with **tilla** to turn the **tilla** object into JSON.
- **Note**
  - The **JSON** object is built-in to JavaScript in all *modern* browsers, just like the **document** object, but you need to make sure you're using a fairly recent version of your favorite browser (IE9+, Safari 5+, Chrome 18+, Firefox 11+, or Opera 11+).

Example

- So now let's create an array of the two objects, and convert the array to JSON.

```
function Pet(type, name, weight, likes) {
		this.type = type;
		this.name = name;
		this.weight = weight;
		this.likes = likes;
		}

window.onload = init;
function init() {
		var pickles = new Pet("cat", "Pickles", 7, ["sleeping", "purring", "eating butter"]);
		console.log(pickles);
		var picklesJSON = JSON.stringify(pickles); console.log(picklesJSON);
		var tilla = new Pet("dog", "Tilla", 25, ["sleeping","eating","walking"]);
		console.log(tilla);
		var tillaJSON = JSON.stringify(tilla);
		console.log(tillaJSON);
		var petsArray = [ pickles, tilla ];
		var petsArrayJSON = JSON.stringify(petsArray); console.log(petsArrayJSON);
}
```

Deserializing an Object

```
function Pet(type, name, weight, likes) {
                this.type = type;
                this.name = name;
                this.weight = weight;
                this.likes = likes;
                }
window.onload = init;
function init() {
                var pickles = new Pet("cat", "Pickles", 7, ["sleeping", "purring", "eating b utter"]);
                var picklesJSON = JSON.stringify(pickles);
                var anotherPickles = JSON.parse(picklesJSON);
                console.log(anotherPickles);
                var tilla = new Pet("dog", "Tilla", 25, ["sleeping","eating","walking"]);
                var petsArray = [ pickles, tilla ];
                var petsArrayJSON = JSON.stringify(petsArray);
                var anotherPetsArray = JSON.parse(petsArrayJSON);
                console.log(anotherPetsArray);
                }
```

- We took **picklesJSON** and we passed it to the **JSON.parse()** method.
- **JSON.parse()** does the opposite of what **JSON.stringify()** does: it takes a piece of data formatted as JSON, and converts it back into a JavaScriptobject (or array).
- It's important to recognize that you get back another object, *not* the same object as our original **pickles** object (now we have two *different* objects,**pickles** and **anotherPickles**).
- So, **JSON.stringify()** takes an object and **serializes** it into JSON format; and **JSON.parse()** takes a piece of data in JSON format and **deserializes** it into a JavaScript object.

+

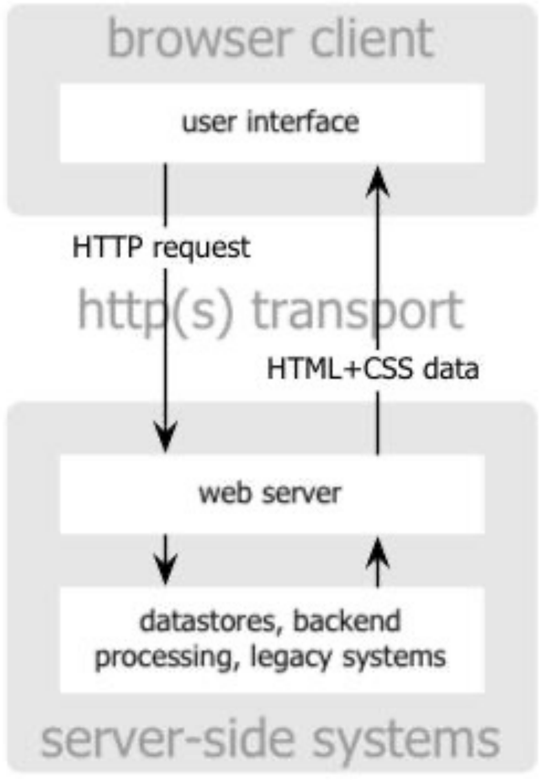
AJAX

What's Ajax

- Ajax is the *buzzword* among web developers
- It stands for ***A****synchronous* ***J****avaScript* ***A****nd* ***X****ML*
- Jesse James Garrett invented this acronym in Feb 2005 to describe its use by Google.
- Ajax isn't a technology
- Ajax is an approach to Web application development that uses client-side scripting to exchange data with the Web server

Classic Model

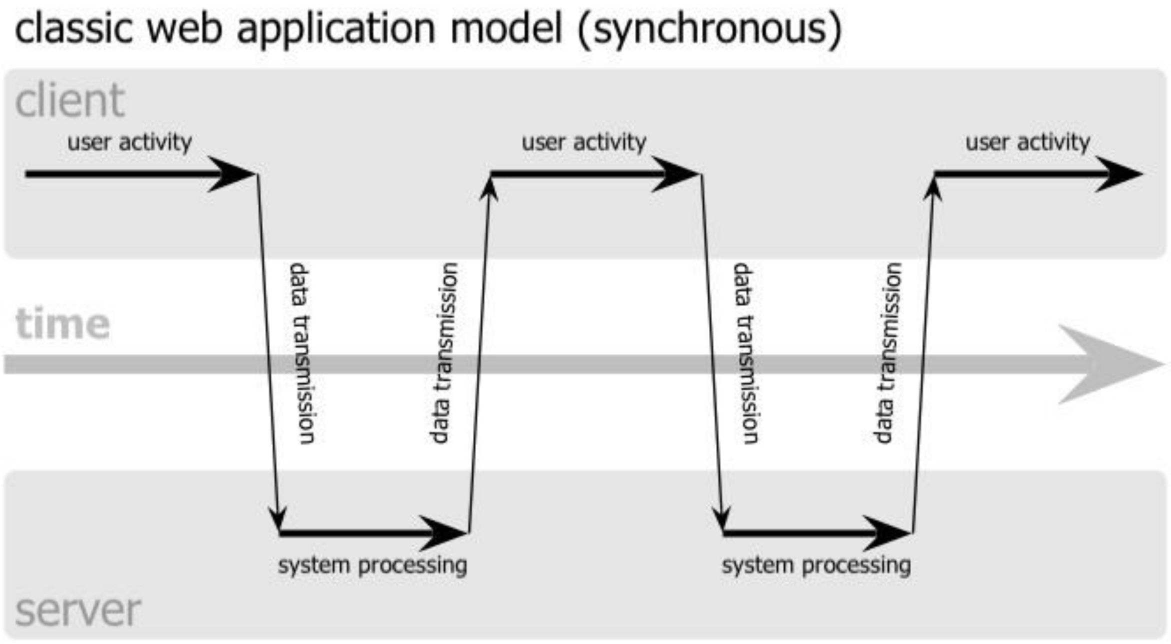
- The classic web application model works like this:
  - Most user actions in the interface trigger an HTTP request back to a web server.
  - The server does some processing — retrieves data, crunches numbers, talks to various legacy systems
  - And then returns an HTML page to the client



classic
web application model

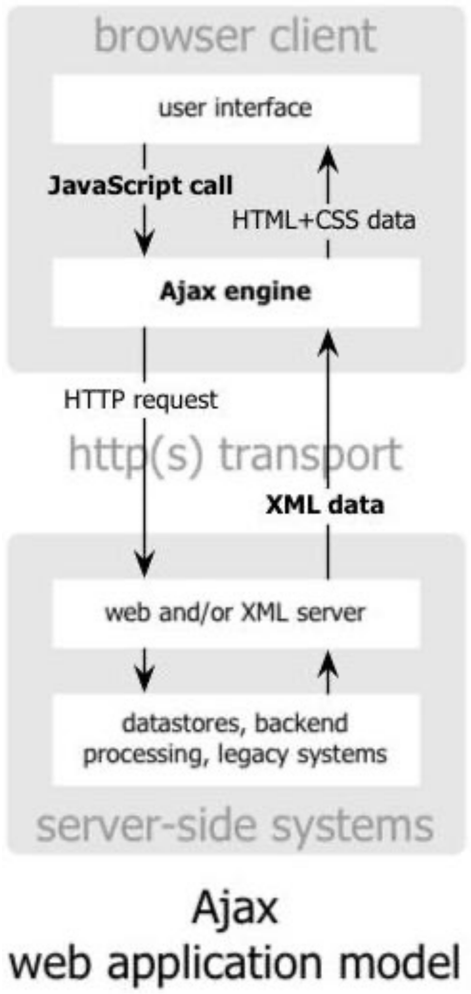
Classic Model

- This approach makes a lot of technical sense, but it doesn't make for a great user experience.
  - At every step in a task, the user waits.
  - The user sees the application go to the server

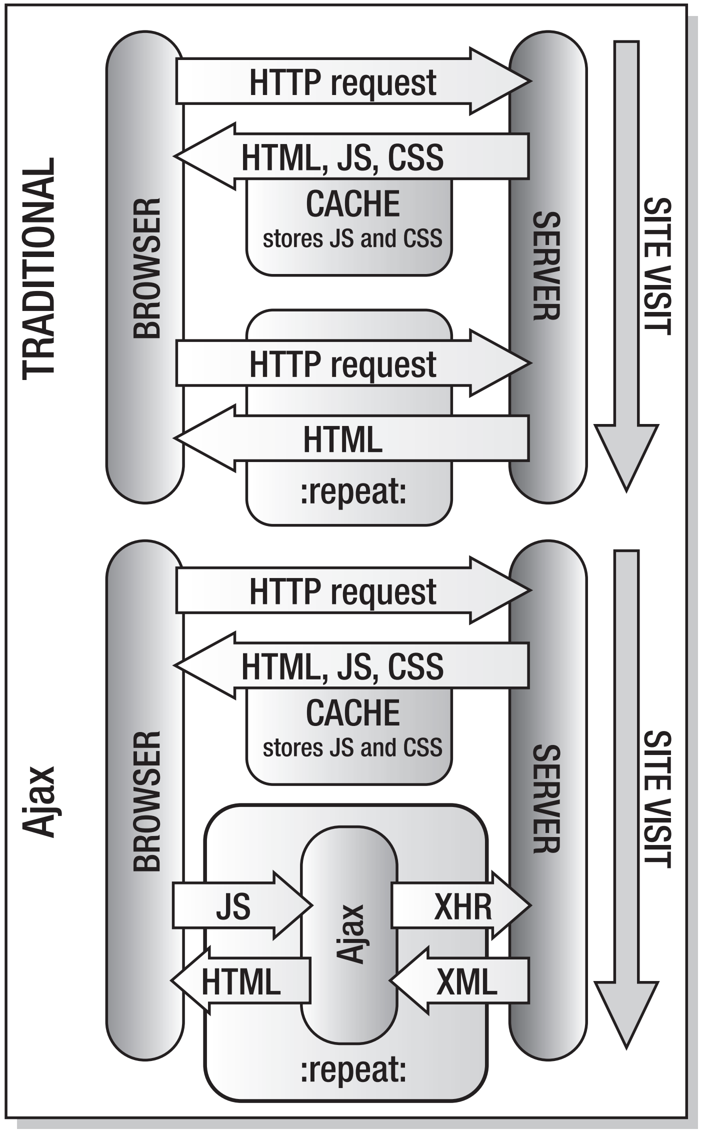


Ajax Model

- An Ajax application eliminates the start-stop-start-stop nature of interaction on the Web
  - It 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 tucked away in a hidden frame.
  - The Ajax engine allows the user's interaction with the application to happen asynchronously, independent of communication with the server



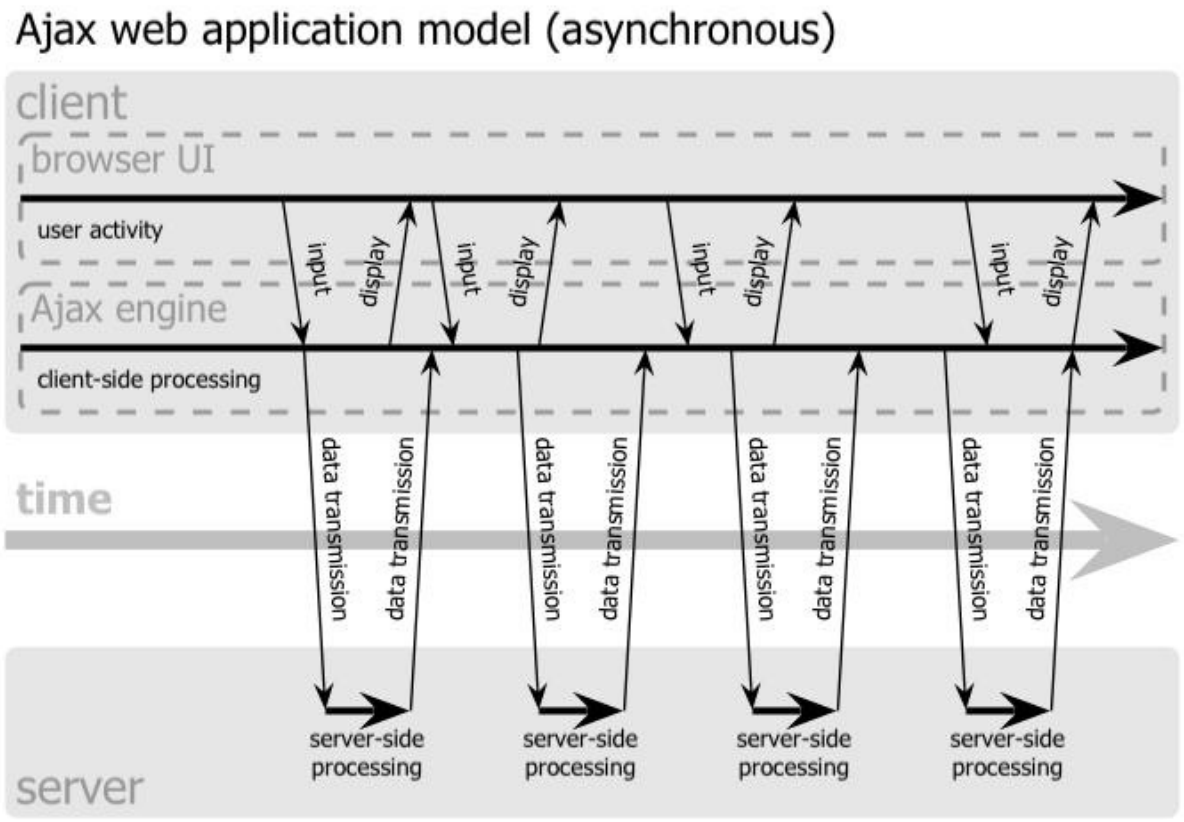
Ajax Model



- 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, and even some navigation — the engine handles on its own.
- If the engine needs something from the server to respond — if it's submitting data for processing, loading additional interface code, or retrieving new data — the engine makes those requests *asynchronously*, usually using XML, without stalling a user's interaction with the application.
- The user is never staring at a blank browser window and an hourglass icon, waiting around for the server to do something.

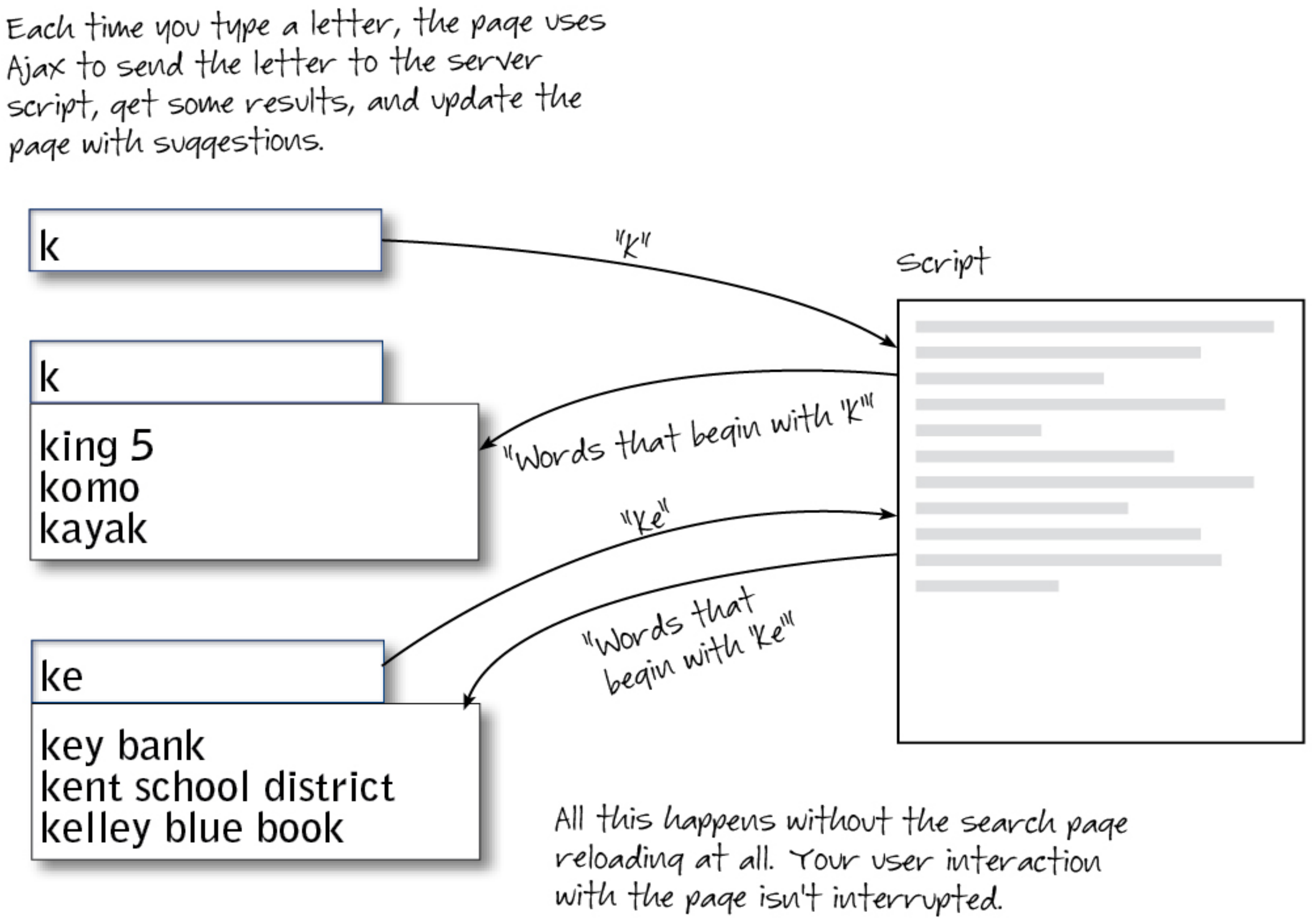
Ajax Model

- Every user action that normally would generate an HTTP request takes the form of a JavaScript call to the Ajax engine



Example, Google

AJAX was made popular in 2005 by Google (with Google Suggest).
Google Suggest is using the XMLHttpRequest object to create a very dynamic Web interface:
When you start typing in Google's search box, a JavaScript sends the letters off to a server, and the server returns a list of suggestions.


Each time you type a letter, the page uses Ajax to send the letter to the server script, get some results, and update the page with suggestions.
k
"k"
Script
k
king 5
komo
kayak
"Words that begin with 'k'"
"ke"
ke
key bank
kent school district
kelley blue book
"Words that begin with 'ke'"
All this happens without the search page reloading at all. Your user interaction with the page isn't interrupted.

XMLHttpRequest

- The kernel of Ajax is the XmlHttpRequest
  - The XMLHttpRequest object allows client-side *JavaScript* to make HTTP requests (both GET and POST) to the server without reloading pages in the browser and without blocking the user
  - This JavaScript object was originally introduced in Internet Explorer 5 by Microsoft, and it is the enabling technology that allows asynchronous requests
  - Despite its name, you can use the XMLHttpRequest object with more than just XML. You can use it to request or send any kind of data.

The XMLHttpRequest object

- The XMLHttpRequest object is the backbone of every Ajax method.
- Each application requires the creation of one of these objects.
- As with most things in web programming, this depends upon the web browser that the client is using because of the different ways in which the object has been implemented in the browsers.
- You can create one of these objects simply using the “new” keyword.

```
 <script type="text/javascript">
      ajaxRequest = new XMLHttpRequest();
</script>
```

*** Old versions of IE do not support the object

```
if (window.XMLHttpRequest) {    // code for modern browsers
   xmlhttp = new XMLHttpRequest();
 } else {      // code for old IE browsers
   xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
}
```

The XMLHttpRequest object (...)

- As with any object in JavaScript (and other programming languages), the XMLHttpRequest object contains various
  - properties and
  - methods.
- Some properties are set after the object is created to specify information to be sent to the server, as well as how to handle the response received from the server.
- Some properties will be updated to hold status information about whether the request finished successfully.
- The methods are used to send the request to the server, and to monitor the progress of the request as it is executed (and to determine if it was completed successfully).

XMLHttpRequest object properties

- readyState
  - An integer from 0. . .4. (0 means the call is uninitialized, 4 means that the call is complete.)
- Onreadystatechange
  - Determines the function called when the objects readyState changes.
- responseText
  - Data returned from the server as a text string (read-only).
- responseXML
  - Data returned from the server as an XML document object (read-only).
- Status
  - HTTP status code returned by the server
- statusText
  - HTTP status phrase returned by the server

+ readyState

- We use the readyState to determine when the request has been completed, and then check the status to see if it executed without an error.
- readyState values:
  - 0 — The object has been created, but the open() method hasn’t been called
  - 1 — The open() method has been called, but the request hasn’t been sent
  - 2 — The request has been sent; headers and status are received and available
  - 3 — A response has been received from the server
  - 4 — The requested data has been fully received

XMLHttpRequest object methods

- open('method', 'URL', asyn)
  Specifies the HTTP method to be used (GET or POST as a string, the target URL, and whether or not the request should be handled asynchronously (asyn should be true or false, if omitted, true is assumed)).
- send(content)
  Sends the data for a POST request and starts the request, if GET is used you should call send(null).
- setRequestHeader('x','y')
  Sets a parameter and value pair x=y and assigns it to the header to be sent with the request.
- getAllResponseHeaders()
  Returns all headers as a string.
- getResponseHeader(x)
  Returns header x as a string.
- abort()
  Stops the current operation.

Sending an aJaX request to a server

- To send a request to a Web server, use the open() and send() methods.
- The open() method takes three arguments.
  - The 1st argument defines which method to use (GET or POST).
  - The second argument specifies the name of the server resource (URL).
  - The third argument specifies if the request should be handled asynchronously.
- The send() method sends the request off to the server.

+

A general skeleton for an Ajax application

```
<script type="text/javascript">
var ajaxRequest = new XMLHttpRequest();
if (ajaxRequest) {   //  if the object was created successfully
     ajaxRequest.onreadystatechange = ajaxResponse;
     ajaxRequest.open("GET", "search.php?query=Bob");
     ajaxRequest.send(null);
  }
function ajaxResponse() {       //This gets called when the readyState changes.
  if (ajaxRequest.readyState != 4) {  return;  }         //  check to see if we're done
else {
    if (ajaxRequest.status == 200) {  //  check to see if successful
               //  process server data here. . .
   } else { alert("Request failed: " + ajaxRequest.statusText); }
    }
}
</script>
```

Testing the Status

- But just knowing that the response is **DONE** isn't quite enough to be sure that we can proceed; we need to make sure that everything went okay.
- Just like when you type a URL into a browser window and send off an HTTP request to get a file, that request could fail if say, the server is down, or the file doesn't exist.
- You might have seen the infamous "500 Internal Server" error, or the "404 Not Found" error before.
- We're sending an HTTP request here too; the server will respond and let us know if everything went okay or if something went wrong.
- This part of the response is put into the request object's **status** property. If the **status** is 200, then we know everything went okay, and we can proceed.

Handling the Response

```
function getPetData() {
        var request = new XMLHttpRequest();
        request.open("GET", "pets.json");
        request.onreadystatechange = function() {
                var div = document.getElementById("pets");
                if (this.readyState == this.DONE && this.status == 200) {
                if (this.responseText != null) { div.innerHTML = this.responseText;}
                else {div.innerHTML = "Error: no data"; }
                }
        };
        request.send(); }
```

- The **callback function** is an *anonymous function*.
  - The end result is that using an anonymous function is exactly the same as using a name, but it bypasses the step where the function is defined with a name.

XMLhttprequest Open—using False

- example

```
xmlhttp.open("GET",url, false);

xmlhttp.send(null);

document.getElementById('test').innerHTML=xmlhttp. responseText;
```

- The third parameter in the open call is "false".
  - This tells the XMLHttpRequest object to wait until the server request is completed before next statement is executed.
  - For small applications and simple server requests, this might be OK.
  - But if the request takes a long time or cannot be served, this might cause your Web application to hang or stop.

XMLhttprequest Open—using true

- By changing the third parameter in the open call to "true", you tell the XMLHttpRequest object to continue the execution after the request to the server has been sent.
- Because you cannot simply start using the response from the server request before you are sure the request has been completed, you need to set the onreadystatechange property of the XMLHttpRequest, to a function (or name of a function) to be executed after completion.
- In this onreadystatechange function, you must test the readyState property before you can use the result of the server call.

- Simply change the code to

```
xmlhttp.onreadystatechange=function() {
   if(xmlhttp.readyState==4) // request is complete
            {document.getElementById('test').innerHTML=xmlhttp. responseText}
   }
   xmlhttp.open("GET",url,true);
   xmlhttp.send(null);
```

+ Simple Example

```
<div id="id01"></div>
<script>
var xmlhttp = new XMLHttpRequest();
var url = "myTutorials.json";

xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
        var myArr = JSON.parse(xmlhttp.responseText);
        myFunction(myArr);
    }
}
xmlhttp.open("GET", url );
xmlhttp.send(null);

function myFunction(arr) {
    var out = "";
    var i;
    for(i = 0; i < arr.length; i++) {
        out += '<a href="' + arr[i].url + '">' + arr[i].display + '</a><br>';
    }
    document.getElementById("id01").innerHTML = out;
}
</script>
```

Simple Example (…)

```
var myArray = [
{
"display": "JavaScript Tutorial",
"url": "http://www.w3schools.com/js/default.asp"
},
{
"display": "HTML Tutorial",
"url": "http://www.w3schools.com/html/default.asp"
},
{
"display": "CSS Tutorial",
"url": "http://www.w3schools.com/css/default.asp"
}
]
```

```
<html>
<head>
 <script type="text/javascript">
          function loadXMLDoc(url) {
          if (window.XMLHttpRequest) {// code for IE7+, Firefox, Chrome, Opera, Safari
                    xmlhttp=new XMLHttpRequest();
                    }
          else {// code for IE6, IE5
                    xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
                    }
          xmlhttp.open("GET",url,false);
          xmlhttp.send(null);
          document.getElementById(‘test’).innerHTML=xmlhttp.responseText;
          }
</script>
</head>
<body>
<div id="test">
 <h2>Clickto let AJAX change this text</h2>
</div>
<button type="button" onclick="loadXMLDoc(‘test1.txt’)">Click Me</button>
<button type="button" onclick="loadXMLDoc(‘test2.txt’)">Click Me</button>

</body>
 </html>
```

Security Concerns

- At first, this kind of call wasn't restricted
- But that meant that if one could inject Javascript into a web page, eg. via a comment form, one could pull live data into a users brower, and thus escape the sandbox
- So now, this method is generally restricted to the same named server…

GET or POST?

- GET is simpler and faster than POST, and can be used in most cases.
- However, always use POST requests when:
  - A cached file is not an option (update a file or database on the server).
  - Sending a large amount of data to the server (POST has no size limitations).
  - Sending user input (which can contain unknown characters), POST is more robust and secure than GET.

+
GET

```
function getAjax(url, success) {
   var xhr = window.XMLHttpRequest ? new XMLHttpRequest() : new ActiveXObject('Microsoft.XMLHTTP');
   xhr.open('GET', url);
   xhr.onreadystatechange = function() {
      if (xhr.readyState>3 && xhr.status==200) success(xhr.responseText);
   };
   xhr.setRequestHeader('X-Requested-With', 'XMLHttpRequest');
   xhr.send();
   return xhr;
}

// example request
getAjax('http://foo.bar/?p1=1&p2=Hello+World', function(data){ console.log(data); });
```

+ POST….

```
function postAjax(url, data, success) {
   var params = typeof data == 'string' ? data : Object.keys(data).map(
         function(k){ return encodeURIComponent(k) + '=' + encodeURIComponent(data[k]) }
      ).join('&');

   var xhr = window.XMLHttpRequest ? new XMLHttpRequest() : new ActiveXObject("Microsoft.XMLHTTP");
   xhr.open('POST', url);
   xhr.onreadystatechange = function() {
      if (xhr.readyState>3 && xhr.status==200) { success(xhr.responseText); }
   };
   xhr.setRequestHeader('X-Requested-With', 'XMLHttpRequest');
   xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
   xhr.send(params);
   return xhr;
}

// example request
postAjax('http://foo.bar/', 'p1=1&p2=Hello+World', function(data){ console.log(data); });

// example request with data object
postAjax('http://foo.bar/', { p1: 1, p2: 'Hello World' }, function(data){ console.log(data); });
```