

Introduction to Ajax

Dott. Ing. Giulio Piancastelli
giulio.piancastelli@unibo.it

Ingegneria Due
ALMA MATER STUDIORUM – Università di Bologna a Cesena

Academic Year 2006/2007

What is Ajax?

- ▶ A cleaning powder
- ▶ A Dutch football team
- ▶ A Greek hero
- ▶ A different approach to web interaction
- ▶ None of the previous
- ▶ All of the previous

History of web interaction (1/2)

- ▶ JavaScript gets released and for the first time developers are able to affect the interaction between the user and the web page (Netscape Navigator 2)
- ▶ Frame are introduced and the web page can be split up into several documents (Netscape Navigator 2)
- ▶ The ability to use JavaScript to control a frame and its contents lets the *hidden frame* technique for client-server interaction emerge
 - ▶ A frame represents a completely separated request to the server
 - ▶ The technique represents the first asynchronous request/response model for web applications

History of web interaction (2/2)

- ▶ DHTML enables developers to alter any part of a loaded page by using JavaScript. Combining it with hidden frames let any part of a page to be refreshed with server information at any time (Internet Explorer 4)
- ▶ Implementation of the DOM standard and the `iframe` HTML element let the *hidden iframe* technique emerge: dynamic creation of an `iframe` to make a request and get a response on the fly (Internet Explorer 5, Netscape 6)
- ▶ The `XMLHttpRequest` object gets introduced as an ActiveX control: an ad-hoc HTTP request that can be controlled from JavaScript independently from the page load/reload cycle (Internet Explorer 5)
- ▶ An `XMLHttpRequest` native JavaScript object gets implemented by virtually every modern web browser

History of Ajax (I)

- ▶ Developers have tried to give users a more interactive and satisfying experience of the web since the days of JavaScript and frames, long before those techniques were called Ajax
- ▶ A technology can be called mature not when some developers play with it, but when big enterprises adopt it for their core business
- ▶ The Ajax turning point is generally considered to be Google Maps, appeared even before the term Ajax was coined

Google Maps (1/2)

The screenshot shows a Google Maps interface. At the top, there are three map style buttons: "Map" (selected), "Satellite", and "Hybrid". On the left side, there is a vertical navigation bar with a compass, a zoom-in (+) button, a zoom-out (-) button, and a scale bar showing 2 miles and 5 kilometers. The main map area displays a coastal region with several towns labeled, including Riccione, Misano Adriatico, Cattolica, and Gradara. A red location pin is placed on the map near Cattolica. A white information popup window is open over the pin, containing the following text:

Ristorante Pizzeria Tre Pini Di Borroni
Ulderico
Via Machiavelli, 15
61011 Gabicce Mare (PU), Italy
0541 954435
marchecitta.it
[Send to phone](#)
Directions: [To here](#) - [From here](#)

At the bottom right of the map, there is an inset map showing the location of the main map area within a larger regional context, with a blue rectangle indicating the current view. The bottom of the map shows copyright information: "© 2006 Google - Map data © 2006 Tele Atlas - [Terms of Use](#)".

- ▶ A scrollable and draggable map broken in a grid of tiles *asynchronously* downloaded from the server *during the normal user workflow*
- ▶ Zooming and zoom level control widget
- ▶ Push pins and active dialogs (with shadows!) used to highlight search results on a map

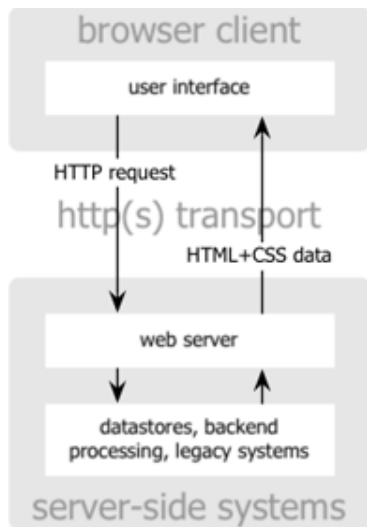
History of Ajax (II)

- ▶ In February 2005, Jesse James Garrett (Adaptive Path) coins the term Ajax, meaning *Asynchronous JavaScript and XML*, and quotes Google Maps and Google Suggest as examples of Ajax applications
- ▶ The Ajax tidal wave raises
 - ▶ because all the Ajax component technologies (HTML, CSS, DOM, XML, JavaScript) are already known and deployed
 - ▶ and because of the Google effect and experience
- ▶ More companies (e.g. Amazon) adopt Ajax on the web
- ▶ Web frameworks (e.g. Ruby on Rails) add Ajax support, and toolkits (e.g. Google Web Toolkit) start to appear

Why is Ajax different?

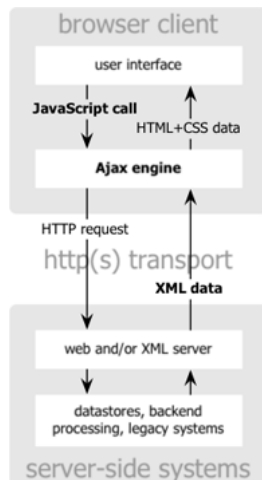
- ▶ Ajax can be viewed as
 - ▶ a set of technologies
 - ▶ a web application architecture
- ▶ More precisely, Ajax can be viewed as a set of relatively old technologies combined to create a new architecture for web applications, meant to increase the page's interactivity, speed, and usability

Classic web application architecture



Ajax web application architecture

- ▶ A new abstraction is introduced, making the interaction engine emerge as a programmable separate entity
- ▶ Requests can be made asynchronously from the user interface



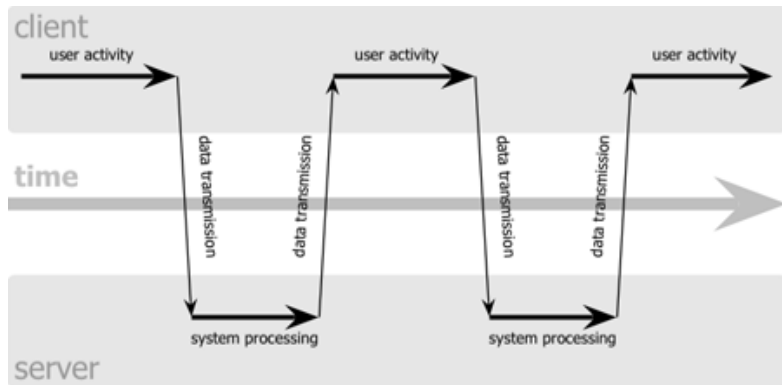
Model View Control

- ▶ Web applications are usually built following the Model View Control design pattern
 - ▶ The Model is represented by the business logic on the server side
 - ▶ The Controller is represented by the application logic, again hosted on the server side
 - ▶ The View is the content of web pages displayed on the client side
- ▶ The introduction of the Ajax engine moves some of the Controller responsibilities on the client side, bringing along some Model representation responsibilities
- ▶ With the introduction of Ajax, the MVC web application pattern gets replicated at a smaller scale

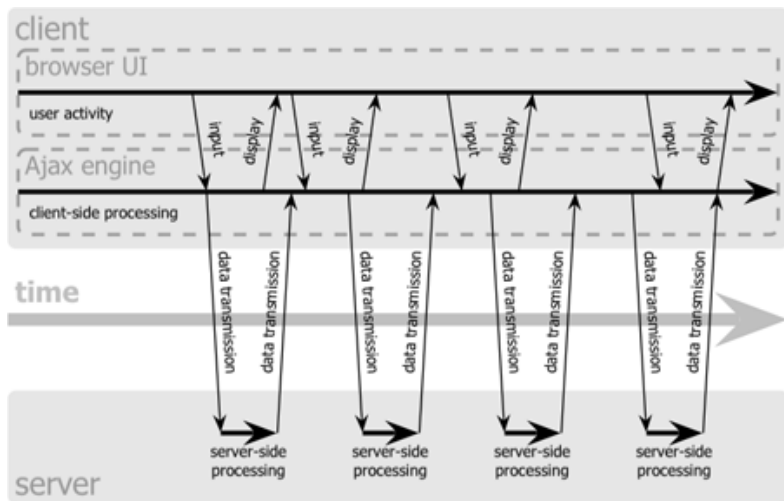
Architecture and interaction model (I)

- ▶ The architecture of an application also dictates the interaction model that the user experiences
- ▶ The web started as a world of document and data sharing, not interactivity in its most meaningful sense
- ▶ In the conventional web application architecture, a page is defined for every event in the system, and each action synchronously returns another full page as its result

Classic web application interaction model



Ajax web application interaction model



Architecture and interaction model (II)

The Ajax web application architecture introduces a new interaction model based around three main characteristics

- ▶ It encompasses *small server side responses*: small requests are made to get only the data needed to update limited portions of a page
- ▶ It is *asynchronous*, so that frequent small requests do not interrupt the user workflow
- ▶ It defines a *fine grained event model* able to trap almost every action a user can do on a page, which may trigger an asynchronous request to the server

From the architecture and the interaction model, some defining principles for the design of Ajax applications can be extracted

- ▶ The browser hosts (part of) an application, not (just) content
- ▶ The server delivers data, not content
- ▶ User interaction with the application can be fluid and continuous

Starting from the defining principles, at least two kinds of Ajax usage can be envisioned

- ▶ The development of self-contained Ajax widgets to be embedded as islands of application-like functionalities into document-like web pages
 - ▶ e.g. Google Suggest, Google Maps
- ▶ The development of a host application, like a desktop application or environment, where application-like and document-like fragments can reside
 - ▶ e.g. Google Mail, ajaxLaunch applications

- ▶ (X)HTML and CSS as standard-based content presentation languages
- ▶ The DOM as a way to perform dynamic display of information and interaction with the user
- ▶ XML as a format for data interchange
- ▶ The XMLHttpRequest object for asynchronous data retrieval from the server side of the application
- ▶ JavaScript as the scripting language gluing these components together

Which technologies do you really need?

- ▶ Developers employed asynchronous techniques way before the XMLHttpRequest object appeared
- ▶ Is XML the only viable format for data interchange?
 - ▶ Does it pay off bringing along companion XML technologies like XPath and XSLT?
 - ▶ Are your data unstructured enough to benefit from the use of a more lightweight interchange format?

- ▶ The page is composed by a `frameset` with a hidden frame which is used for communication between the client and server side of the application
- ▶ The *hidden frame* technique follows a four-step pattern
 1. A JavaScript call is made to the hidden frame, as a consequence of an user's action which requires additional data from the server
 2. A request is made to the server
 3. The response is received from the server, under the form of a web page, since the application deals with frames
 4. The received web page in the hidden frame calls a JavaScript function in the visible frame to transfer the data there, and let the application decide what to do with the data

Intermission: HTTP request

- ▶ Unlike RPC, requests in HTTP are directed to resources using a generic interface with standard semantics that can be interpreted by intermediaries almost as well as by the machines that originated services
- ▶ The most important methods in the generic HTTP interface and their semantics are
 - GET** to retrieve a representation of the resource in an idempotent way
 - POST** to create a new subordinate entity of the specified resource using the content sent in the request
 - PUT** to create or modify the specified resource using the content sent in the request
 - DELETE** specifying that the resource must be deleted

GET requests with hidden frame (1/4)

- ▶ Start with a frameset

```
<frameset rows="100%,0" frameborder="0">  
  <frame name="displayFrame" noresize="noresize"  
    src="get_display_frame.html" />  
  <frame name="hiddenFrame" noresize="noresize"  
    src="about:blank" />  
</frameset>
```

- ▶ The rows, frameborder, noresize attributes help hiding the frame and the framing nature of the page
- ▶ The hidden frame starts blank

GET requests with hidden frame (2/4)

- ▶ In the visible frame, given a stock name, its current value will be displayed

```
<p>Stock Name:  
  <input type="text" id="stockName" value="" /></p>  
<p><input type="button" value="Get Stock Value"  
  id="stockValueButton" /></p>  
<div id="stockValue"></div>
```

- ▶ From the stockName element the stock name is taken
- ▶ To the stockValueButton element a JavaScript function calling the hidden frame will be associated, separating view from control
- ▶ In the stockValue element the stock value will be displayed

GET requests with hidden frame (3/4)

Here are the JavaScript functions for the visible frame

```
function requestStockValue() {
    var stockName = document.getElementById('stockName').value
    top.frames['hiddenFrame'].location =
        'getStockValue.jsp?stock=' + stockName
}
function displayStockValue(value) {
    var divStockValue =
        document.getElementById('stockValue')
    divStockValue.innerHTML = value
}
window.onload = function() {
    document.getElementById('stockValueButton').onclick =
        requestStockValue
}
```

GET requests with hidden frame (4/4)

- ▶ On the server side, the JSP page will return to the hidden frame an entire web page containing the stock value in an HTML element

```
<div id="stockValue"><%= value %></div>
```

- ▶ A JavaScript function is automatically triggered when the page gets loaded in the hidden frame, to return sensible data to the visible frame

```
window.onload = function() {  
    var stockValue =  
        document.getElementById('stockValue').innerHTML  
    top.frames['displayFrame'].displayStockValue(stockValue)  
}
```

POST requests with hidden frame (1/3)

- ▶ The frameset does not change, but the visible frame now contains a form to submit data

```
<form method="POST" action="saveStock.jsp"
      target="hiddenFrame">
  <p>Stock Name:
    <input type="text" name="stockName" value=""/></p>
  <p><input type="submit" value="Save Stock"/></p>
</form>
<div id="stockStatus"></div>
```

- ▶ The stockStatus element will contain information on the server status after the request has been accepted and the action has been performed

POST requests with hidden frame (2/3)

- ▶ In the visible frame, only one JavaScript function needs to be defined

```
function saveResult(message) {  
    var status = document.getElementById('stockStatus')  
    status.innerHTML = 'Request completed: ' + message  
}
```

- ▶ The `saveResult` function will be invoked by the page returned in the hidden frame, receiving data from the response and following the usual four-step pattern of the hidden frame technique

POST requests with hidden frame (3/3)

- ▶ Provided that the JSP page on the server side saves the status of the executed request in the status variable, only one JavaScript function needs to be defined

```
window.onload = function() {  
    top.frames['displayFrame'].saveResult('<%= status %>')  
}
```

- ▶ It will be automatically executed after page loading
- ▶ The saveResult function needs a string as an argument, so the JSP inline invocation needs to be put between quotes
- ▶ The page body can be left empty

Ajax with `iframe` elements

- ▶ An `iframe` element is the same as a `frame` that can be placed inside a non `frameset` HTML page
- ▶ The `iframe` technique can be applied to pages not originally created as a `frameset`, making it better suited to incremental addition of functionality
- ▶ An `iframe` element can even be created on the fly by some JavaScript code, allowing better separation between HTML and dynamic Ajax enhancements
- ▶ Note that `iframe` elements can be used and accessed in the same way as regular frames

GET requests with iframe elements

- ▶ No more frameset
- ▶ The displayed page contains a hidden iframe

```
<iframe src="about:blank" name="hiddenFrame"
        width="0" height="0" frameborder="0">
</iframe>
```
- ▶ In the document loaded by the hidden iframe, the JavaScript function that calls back the visible page need not to access another frame, but only its parent element

```
parent.displayStockValue(stockValue)
```

Dynamic creation of iframe elements (1/2)

- ▶ An iframe element can be easily created on the fly using the DOM JavaScript API

```
function createIframe() {  
    var iframeElement = document.createElement('iframe')  
    iframeElement.name = 'hiddenFrame'  
    iframeElement.id = 'hiddenFrame'  
    iframeElement.width = 0  
    iframeElement.height = 0  
    iframeElement.frameBorder = 0  
    document.body.appendChild(iframeElement)  
    return frames['hiddenFrame']  
}
```

- ▶ The function returns a reference to the iframe element just created

Dynamic creation of iframe elements (2/2)

- ▶ Some browsers may happen not to immediately recognized the inserted iframe and allow requests to be sent, so a little timeout trick could be needed

```
var iframe = null
function requestStockValue() {
  if (!iframe) {
    iframe = createIframe()
    setTimeout(requestStockValue, 10)
    return
  }
  // use the iframe just created
}
```

- ▶ The timeout let the requestStockValue function be called again after a time interval of 10 milliseconds if the iframe is not recognized

POST requests with `iframe` elements (1/4)

- ▶ Not every browser let the target of a form be set to a dynamically created `iframe`
- ▶ To accomplish a POST request with a hidden `iframe`, a different approach has to be employed
 1. load a page that contains an empty form into the hidden `iframe`
 2. populate that form with data from the visible form
 3. submit the hidden form instead of the visible form
- ▶ The visible form submission has to be cancelled and the information it contains needs to be forwarded to the form in the hidden `iframe`

POST requests with iframe elements (2/4)

- ▶ Create the hidden iframe and load it in a page that contains a form

```
var iframe = null
function checkIframe() {
  if (!iframe)
    iframe = createIframe()
    setTimeout(function() {
      iframe.location = 'proxy_form.html'
    }, 10)
}
```

- ▶ Intercept the visible form's submission and cancel it

```
window.onload = function() {
  var stockForm = document.getElementById('stockForm')
  stockForm.onsubmit = function() {
    checkIframe(); return false
  }
}
```

POST requests with `iframe` elements (3/4)

- ▶ The new hidden form is initially empty

```
<form method="POST"></form>
```

- ▶ When loaded into the hidden `iframe`, the page containing the proxy form asks the visible page to transfer data from the original form to itself

```
window.onload = function() { parent.setForm() }
```

- ▶ The hidden form gets populated and submitted
- ▶ In the document loaded by the hidden `iframe`, the JavaScript function that calls back the visible page need not to access another frame, but only its parent element

```
parent.saveResult('<%= status %>')
```

POST requests with iframe elements (4/4)

```
function setForm() {  
  var hiddenForm = iframe.document.forms[0]  
  var form = document.forms[0]  
  for (var i = 0; i < form.elements.length; i++) {  
    var hiddenInput = iframe.document.createElement('input')  
    hiddenInput.type = 'hidden'  
    hiddenInput.name = form.elements[i].name  
    hiddenInput.value = form.elements[i].value  
    hiddenForm.appendChild(hiddenInput)  
  }  
  hiddenForm.action = form.action  
  hiddenForm.submit()  
}
```

Advantages of hidden frames

- ▶ Hidden frames maintain browser history and thus enable users to use the Back and Forward buttons effectively
- ▶ The browser keeps track of the requests made through frames, and the Back and Forward buttons move through history of frames whereas the main page of the application does not change
- ▶ Be careful with `iframe` elements: depending on how they are created and the browser used, `iframe` history may or may not be kept

Disadvantages of hidden frames

- ▶ The application relies on the proper page being correctly returned to the hidden frames
- ▶ There is very little information about what happens in the hidden frames
 - ▶ No notification of problems loading the page
 - ▶ Timeout techniques can be employed, but they are just a workaround
 - ▶ A developer can not control the HTTP request nor the HTTP response being returned

Ajax with XMLHttpRequest

- ▶ The XMLHttpRequest or XMLHttpRequest object enables developers to initiate a HTTP transaction from anywhere in an application
- ▶ This technique follows the callback pattern
 1. An XMLHttpRequest object is created and the request is initialized
 2. JavaScript asks the browser to perform the actual HTTP request towards the server
 3. The browser receives the HTTP response from the server and calls back a previously set XMLHttpRequest handler to manage data and information from the response

Creating a XMLHttpRequest (1/3)

- ▶ The two different implementations as a JavaScript native object or as an ActiveX control must be taken into account
- ▶ For non-Microsoft browsers, creation is quite simple

```
var request = new XMLHttpRequest()
```
- ▶ For Microsoft browsers, it could be as simple...

```
var request = new ActiveXObject('Microsoft.XMLHTTP')
```
- ▶ ...but that line creates only the first version of the object: what to do if a more recent version has to be used?

Creating a XMLHttpRequest (2/3)

The only way to determine the best Microsoft object version to use is to try and create each one

```
var VERSIONS = [ "MSXML2.XMLHttp.5.0", "MSXML2.XMLHttp.4.0",  
                "MSXML2.XMLHttp.3.0", "MSXML2.XMLHttp",  
                "Microsoft.XMLHttp" ]  
for (var i = 0; i < VERSIONS.length; i++) {  
    try {  
        var request = new ActiveXObject(VERSIONS[i])  
        return request  
    } catch (error) {  
        // do nothing  
    }  
}
```

Creating a XMLHttpRequest (3/3)

You also need to distinguish between the two implementations to know which is the correct one to create in the context of the browser in use

```
function createXmlHttpRequest() {
  if (typeof XMLHttpRequest != 'undefined')
    return new XMLHttpRequest()
  else if (window.ActiveXObject) {
    // create and return the appropriate Microsoft object
  }
  throw new Error('XMLHttpRequest object could not be created.')
}
```

Methods of XMLHttpRequest (1/2)

`getAllResponseHeaders()` returns all the HTTP response headers as key-value string pairs

`getResponseHeader(header)` takes the name of the specified response header as a string and returns its value as a string

`open(method, url, async, user, password)` initializes the method HTTP request directed to `url`

`async` is an optional boolean parameter, defaulted to true, to specify if the request should be performed in an asynchronous way

Methods of XMLHttpRequest (2/2)

`send(content)` asks the browser to perform the HTTP request and immediately returns if the request is to be asynchronously made

`content` is a mandatory argument that can be a DOM document instance, a string, or a stream, and its content is sent as the request body

`abort()` stops the current request

Properties of XMLHttpRequest

`onreadystatechange` is the event handler that fires its assigned callback function at every state change of the request

`readyState` is an integer representing the state of the request

`responseText` is the body of the server response, represented as a string

`responseXML` is the body of the server response, represented as an XML DOM document object

`status` is the HTTP response numeric status received from the server

`statusText` is the HTTP response textual status received from the server

XMLHttpRequest ready states

The `readyState` property changes its value as the HTTP transaction is performed, the request is set, and the response is received

0. When the `XMLHttpRequest` object is created, its `readyState` property is set to 0 (uninitialized)
1. When the `open` method is called, the request gets initialized and `readyState` is set to 1 (loading)
2. Immediately after the request has been sent using the `send` method, `readyState` changes to 2 (loaded)
3. When the browser starts to receive data from the HTTP response, `readyState` passes to the value 3 (interactive)
4. When all response data has been received and the connection with the server has been closed, `readyState` changes to 4 (complete)

Because of differences in browser implementations, the only reliable ready states for cross browser development are 0, 1, and 4

Intermission: HTTP response status codes

- ▶ The HTTP response status code is analogous to a summary of the response, and lets the client know the basic outcome of the server's attempt to fulfill the request
- ▶ Status codes are grouped into ranges, and the most important are the following

Successful (200-299) for example, 200 OK indicates that the request has succeeded

Client error (400-499) for example, 404 Not Found indicates that the resource cannot be found

Server error (500-599) for example, 501 Not Implemented indicates that the server does not support the method used in the HTTP request

GET requests with XMLHttpRequest (1/3)

- ▶ A request must be created and opened towards the appropriate server side resource

```
var stockName = document.getElementById('stockName').value
var request = createXmlHttpRequest()
request.open('get', 'getStockValue.jsp?stock=' + stockName)
```

- ▶ An `onreadystatechange` handler must be defined, which will be called back by the browser on each ready state transition
 - ▶ The handler needs to check the `readyState` to be sure to perform its action only when the response has been actually received. . .
 - ▶ . . . and it needs to check the response status code in order to provide some support if something goes wrong

GET requests with XMLHttpRequest (2/3)

- ▶ The handler can be created inline as an anonymous function

```
request.onreadystatechange = function() {  
  if (request.readyState == 4)  
    if (request.status == 200)  
      displayStockValue(request.responseText)  
    else  
      displayStockValue('An error occurred: ' +  
                          request.statusText)  
}
```

- ▶ Finally, the request must be sent, without any content since it is of the GET sort

```
request.send(null)
```

GET requests with XMLHttpRequest (3/3)

- ▶ The server side JSP page returns just the stock value as a simple plain text document

```
<%@page contentType="text/plain" %>  
<%  
// calculate the stock value and store it in value...  
out.println(value);  
%>
```

- ▶ No JavaScript code is required outside the main page

Pssst... a word about browser caching

- ▶ Some browsers tend to cache certain resources to improve the speed of displaying and downloading sites
- ▶ When those browsers deal with repeated GET requests to the same page, they may happen to present the user the same old cached response instead of asking anew information to the server
- ▶ To avoid inconvenience, use the appropriate HTTP header on any data being sent from the server...

`Cache-Control: no-cache`

- ▶ ...and another header to maintain backward compatibility with clients implementing HTTP/1.0 only

`Pragma: no-cache`

POST requests with XMLHttpRequest (1/3)

- ▶ The form submission must be prevented and its data must be assembled into the body of the JavaScript request represented by a XMLHttpRequest object

```
document.getElementById('stockForm').onsubmit = function() {  
    sendRequest()  
    return false  
}
```

- ▶ The data for a POST request must be sent in the following format, similar to a query string...

```
name1=value1&name2=value2...
```

- ▶ ...but both the name and value of each parameter must be encoded in order to avoid data loss during transmission

POST requests with XMLHttpRequest (2/3)

- ▶ To assemble the JavaScript request, the following function can be used

```
function getRequestBody(form) {  
    var params = new Array()  
    for (var i = 0; i < form.elements.length; i++) {  
        var param = encodeURIComponent(form.elements[i].name)  
            + '='  
            + encodeURIComponent(form.elements[i].value)  
        params.push(param)  
    }  
    return params.join('&')  
}
```

- ▶ The JavaScript built-in function `encodeURIComponent` is used to perform parameter encoding
- ▶ Parameters are stored in an array and later joined in a string with a predefined separator

POST requests with XMLHttpRequest (3/3)

- ▶ A request must be created with the appropriate target

```
var form = document.forms[0]
var body = getRequestBody(form)
var request = createXmlHttpRequest()
request.open('post', form.action)
```

- ▶ Since it is a POST request, the content type must be appropriately set up

```
request.setRequestHeader('Content-Type',
    'application/x-www-form-urlencoded')
```

- ▶ The structure of the onreadystatechange handler is identical to the GET request example

- ▶ The request must be sent with its body

```
request.send(body)
```

- ▶ The server side JSP page changes in the same way as the GET request example

Advantages of XMLHttpRequest

- ▶ Cleaner code intent
- ▶ Better separation of JavaScript code playing the role of the application controller
- ▶ Complete access to HTTP request and response properties enabling a better error handling
- ▶ Freedom to use a XMLHttpRequest or XMLHttpRequest object anywhere in the code

Disadvantages of XMLHttpRequest

- ▶ For security reasons, only URLs on the same server and port as the page that includes the call to XMLHttpRequest can be loaded
- ▶ There is no browser history record of the calls that are made using XMLHttpRequest
- ▶ There is a supplemental unwanted dependency on ActiveX controls in Microsoft browsers which could have been disabled by the user for security reasons

Which is the better technique to use?

- ▶ Given known advantages and disadvantages of both, developers should use the technique that better suites their application
- ▶ Do not forget application design principles
 - ▶ Be consistent in user interface choices
 - ▶ Follow established conventions in interaction
 - ▶ Avoid unnecessary and distracting elements
 - ▶ Consider accessibility issues
 - ▶ Design with the user in mind before anything else
- ▶ Bottom line: the greatest applications like Google Mail or Google Maps use a mix of both techniques to make a truly usable interface and provide a better user experience

What about PUT and DELETE requests?

The PUT and DELETE HTTP requests are the stepchildren of the web

- ▶ Forms do not support PUT and DELETE methods
- ▶ Some platforms, notably J2ME, do not even implement PUT and DELETE requests in their HTTP library
- ▶ Incomplete implementations of XMLHttpRequest are scattered around browsers without support for PUT and DELETE requests

Payload format

- ▶ The data returned by a HTTP response, commonly known as the *payload*, can be of many sorts
 - ▶ HTML or XML fragments
 - ▶ Scripts to be executed by the client
 - ▶ Arbitrary data in a variety of formats, starting from simple plain text
- ▶ The web has lately seen the birth of some lightweight formats to represent arbitrary data avoiding use of XML, and the creation of software libraries for manipulating data in those formats

Why use XML?

- ▶ Despite the acronym buzzword, the use of XML as a data format is not needed to implement Ajax techniques
- ▶ As with any other technology, XML should be adopted whenever developers think it is necessary for the application
 - ▶ Do they want to define custom data vocabularies?
 - ▶ Do they need human readable payloads?
 - ▶ Do they want to introduce data validation?
 - ▶ Do they want to take advantage of companion technologies like XSLT and XPath, or have to work with XML based technologies such as feeds or web services?
- ▶ Do not forget the *Keep It Simple, Stupid* (KISS) principle

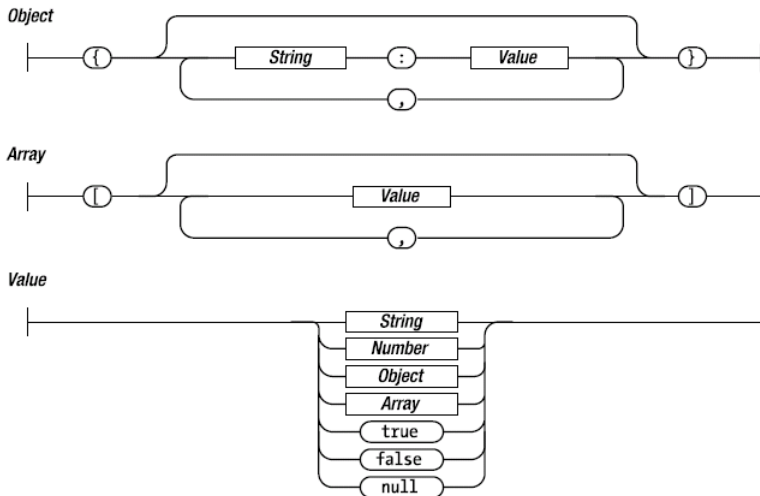
- ▶ If the HTTP response returns XML as its payload, the `responseXML` property of the `XMLHttpRequest` object will contain the data as a DOM document object
- ▶ The object stored in the `responseXML` property can be accessed and manipulated using the standard DOM API, also used with HTML documents
- ▶ From version 1.6 of the JavaScript language, the E4X extension has been introduced, allowing manipulation of XML documents as native data objects
 - ▶ Create an XML object using a string representation of the document, then easily manipulate and access its elements
 - ▶ Unfortunately, only on SeaMonkey (1.0 and above) and Firefox (1.5 and above)

- ▶ As opposed to the heavyweight XML format, Douglas Crockford proposed a new data format called JavaScript Object Notation¹
- ▶ The format is based on the JavaScript notation for array and object literals

```
var stockNames = ['Bnl', 'Fiat', 'Juventus FC']
var stock = {'name':'Fastweb', 'value':38.28}
```
- ▶ JSON syntax is really nothing more than the mixture of object and array literals to store data

¹See <http://json.org> for full details on JSON

JSON syntax



Manipulating JSON data (1/2)

- ▶ To manipulate JSON data on the client side, it is necessary to transform its string representation into an object

- ▶ Given the JSON string...

```
var s = '{"colors":["red", "blue", "gray"], "doors":[2, 4]}'
```

- ▶ ...transforming it into an object is as simple as...

```
var carInfo = eval("(" + s + ")")
```

- ▶ ...then the object can be used as usual

```
carInfo.colors[1] // returns blue
```

```
carInfo.doors[0] // returns 2
```

Manipulating JSON data (2/2)

- ▶ Extra parentheses around any JSON string must be included before passing it to `eval`, because they help indicating that the code between curly braces is an expression to be evaluated, not a statement to be run
- ▶ But the use of `eval` can be a huge security risk: libraries are available to parse and convert only JSON code into objects instead of evaluating arbitrary JavaScript code
- ▶ On the server side, there is plenty of libraries for the most popular languages, starting with Java

```
<?xml version="1.0"?>
<stocks>
  <stock>
    <name>Bnl</name>
    <value>3.21</value>
  </stock>
  <stock>
    <name>Unipol</name>
    <value>2.40</value>
  </stock>
</stocks>
```

```
{ 'stocks' : [
  { 'name': 'Bnl',
    'value': 3.21 },
  { 'name': 'Unipol',
    'value': 2.40 }
]
```

Towards Ajax design patterns

- ▶ Design patterns describe programming techniques known to be successful at solving common problems
- ▶ As Ajax emerges, developers learn more about what sorts of design work, and need ways of documenting this information and talking about it
- ▶ Design patterns are an excellent means of knowledge representation: a concise way to represent the knowledge embodied in the Ajax applications living currently on the web
- ▶ The aim is to discover best practices by investigating how developers have successfully traded off conflicting design principles, delivering usability in the face of constraints

- ▶ The techniques described under the term Ajax have already been extensively used in the past, giving rise to several Ajax patterns that solve specific problems
- ▶ The hidden frame technique and the asynchronous XMLHttpRequest call are two of these patterns
- ▶ Various pattern classifications exist
- ▶ The community has to generate collective wisdom around specific patterns, and individuals need to decide whether and how to implement a given pattern

Pattern example: periodic refresh (1/2)

- ▶ It belongs to the Browser-Server Dialogue category of Programming Patterns
- ▶ Problem: how can the application keep users informed of changes occurring on the server?
- ▶ Forces
 - ▶ The state of many web applications is inherently volatile because changes can come from numerous sources
 - ▶ HTTP requests can only emerge from the client
- ▶ Solution: the browser periodically issues a XMLHttpRequest call to gain new information

Pattern example: periodic refresh (2/2)

- ▶ Set an interval for the periodic request

```
var interval = 10000 // milliseconds
```

- ▶ In the `onreadystatechange` handler, retrigger the main function containing it after the periodic interval

```
function requestStockValue() {  
    request.onreadystatechange = function() {  
        if (request.readyState == 4) {  
            // do something when status is 200...  
            setTimeout(requestStockValue, interval)  
        }  
    }  
}
```

- ▶ Start the main function with periodic requests in the `window.onload` listener

Pattern example: one second spotlight (1/2)

- ▶ It belongs to the Visual Effects category of Functionality and Usability Patterns
- ▶ Problem: how can you direct the user's attention to spots on the page?
- ▶ Forces
 - ▶ To ensure the user is working with current data, the browser display must be frequently updated
 - ▶ The screen can get cluttered with a lot of information, much of which might be changing at any time
 - ▶ While human vision is good at spotting changes, it is easy to miss a sudden change, especially if it is a subtle one
- ▶ Solution: when a display element undergoes a significant change, dynamically increase its brightness for a second

Pattern example: one second spotlight (2/2)

To fade an element the following algorithm may be used

1. Remember the element's current color setting
2. Set `element.style.color` to a bright setting
3. Then, every 100 milliseconds...
 - 3.1 Fade the element a bit. More precisely, drop color by 10% of the bright setting. This applies individually to each color component (R, G, B)
 - 3.2 Check if 1000 milliseconds has already passed. If so, set the element back to its original color setting (it should already be about that anyway)

Towards Ajax frameworks

- ▶ Frequently used Ajax techniques and design patterns get usually provided by libraries, toolkits, and frameworks which make them more easily accessible in production ready code
- ▶ Ajax support in tools gets quite layered as more and more details are shielded from the developer
- ▶ Three main categories can be envisioned
 - ▶ Remoting toolkits
 - ▶ UI toolkits
 - ▶ Web frameworks with Ajax support

Ajax frameworks (1/2)

Ajaxian Web Frameworks

Rails, Tapestry, WebWork, ASP.NET, ...

UI Toolkit

Dojo, SmartClient, Backbase, ...

Remoting Toolkit

DWR, JSON-RPC, dojo.io.bind()

XMLHttpRequest

iframe

...

JavaScript
Utilities
& Tools

Ajax frameworks (2/2)

- ▶ Remoting toolkits provide wrappers around XMLHttpRequest to make it more usable, adding support for error handling and automatic fallback for older browsers
- ▶ UI toolkits provide easy access to visual effects and to smart and richer widget components
- ▶ Web frameworks can be Ajax-aware in different ways: incorporating libraries and toolkits, with automatic code generation, or exploiting component technology

Using Prototype: periodic refresh

- ▶ Prototype² is a JavaScript library providing DOM extensions and implementations for many fundamental low-level Ajax patterns
- ▶ The `Ajax.PeriodicalUpdater` object provides a complete realization of the periodic refresh pattern, which the developer only needs to properly configure

```
new Ajax.PeriodicalUpdater(  
    'stockValue',  
    'getStockValue.jsp?stock=' + stockName,  
    { method: 'get', frequency: interval, decay: 0 }  
)
```

- ▶ No need to act directly on `XMLHttpRequest` and its `onreadystatechange` property

²Home page at <http://www.prototypejs.org>

Using Script.aculo.us: one second spotlight



- ▶ Script.aculo.us³ is a JavaScript library built on top of Prototype, adding more Ajax goodies such as visual effects
- ▶ The one second spotlight pattern is implemented by a single object taking as a constructor parameter the id of the element to be highlighted

```
new Effect.Highlight('stockValue')
```

- ▶ Periodic spotlight can be realized in combination with Prototype's `Ajax.Updater` object and the `setInterval` standard JavaScript function

³Download it from <http://script.aculo.us>

-  Jesse James Garrett, *Ajax: a New Approach to Web Applications*, Adaptive Path, 2005, available at <http://www.adaptivepath.com/publications/essays/archives/000385.php>
-  Nicholas Zakas, Jeremy McPeak, Joe Fawcett, *Professional Ajax*, Wiley Publishing, 2006
-  Justin Gehtland, Ben Galbraith, Dion Almaer, *Pragmatic Ajax*, Pragmatic Bookshelf, 2006
-  Brett McLaughlin, *Head Rush Ajax*, O'Reilly, 2006
-  Ryan Asleson, Nathaniel Schutta, *Foundations of Ajax*, Apress, 2005
-  Michael Mahemoff, *Ajax Design Patterns*, O'Reilly, 2006

-  Roy Fielding, James Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, Tim Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, Internet RFC 2616, June 1999, available at <http://www.w3.org/Protocols/rfc2616/rfc2616.html>
-  Roy Fielding, *Architectural Styles and the Design of Network-based Software Architecture* Ph.D. Thesis, University of California, Irvine, 2000