Unit-5

INTRODUCTION TO AJAX and WEB SERVICES

INTRODUCTION TO AJAX

The underlying technologies behind classic Web applications (HTML) are simple and straight forward. The classic web pages has very little intelligence and lack dynamic and interactive behaviors. Changes in today's web pages are brought by AJAX (Asynchronous JavaScript and XML)

Ajax refers to a set of technologies and techniques that allow web pages be interactive like desktop applications.

AJAX is a new technique for creating better, faster, and more interactive web applications with the help of XML, HTML, CSS, and Java Script. Ajax uses XHTML for content, CSS for presentation, along with Document Object Model and JavaScript for dynamic content display. Conventional web applications transmit information to and from the sever using synchronous requests. It means the user fill out a form, hit submit, and get directed to a new page with new information from the server. With AJAX, when the user hit submit, JavaScript will make a request to the server, interpret the results, and update the current screen. The user would never know that anything was even transmitted to the server.

XML is commonly used as the format for receiving server data. AJAX is a web browser technology independent of web server software. A user can continue to use the application while the client program requests information from the server in the background. In AJAX, clicking is not required; mouse movement is a sufficient event trigger. It is a data-driven technology. AJAX cannot work independently. It is used in combination with other technologies to create interactive webpages. The technologies that support AJAX are: JavaScript, DOM, CSS and XMLHttpRequest.

AJAX is based on the following open standards:

- Browser-based presentation using HTML and Cascading Style Sheets (CSS).

- Data is stored in XML format and fetched from the server.
- Behind-the-scenes data fetching is done using XMLHttpRequest objects in the browser.
- JavaScript to make everything happen.

Asynchronous nature of AJAX

Asynchronous in AJAX means that the script will send a request to the server, and continue the execution without waiting for the reply. As soon as reply is received a browser event is fired, which in turn allows the script to execute associated actions. The client and the server are asynchronous.

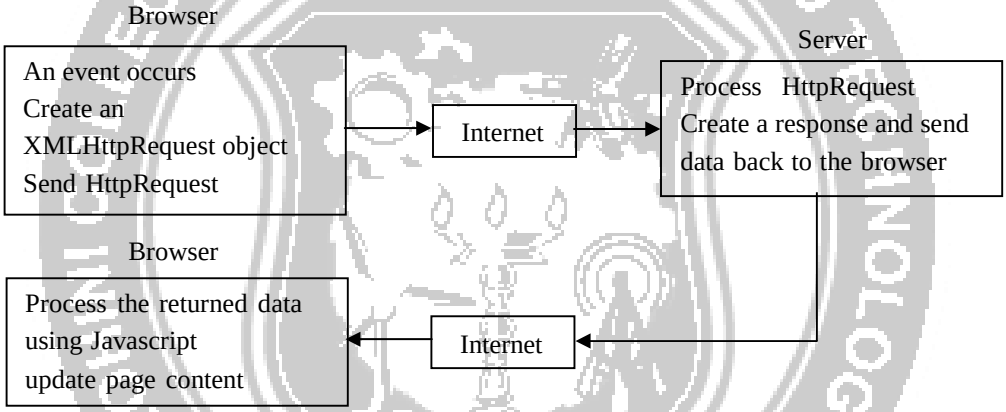
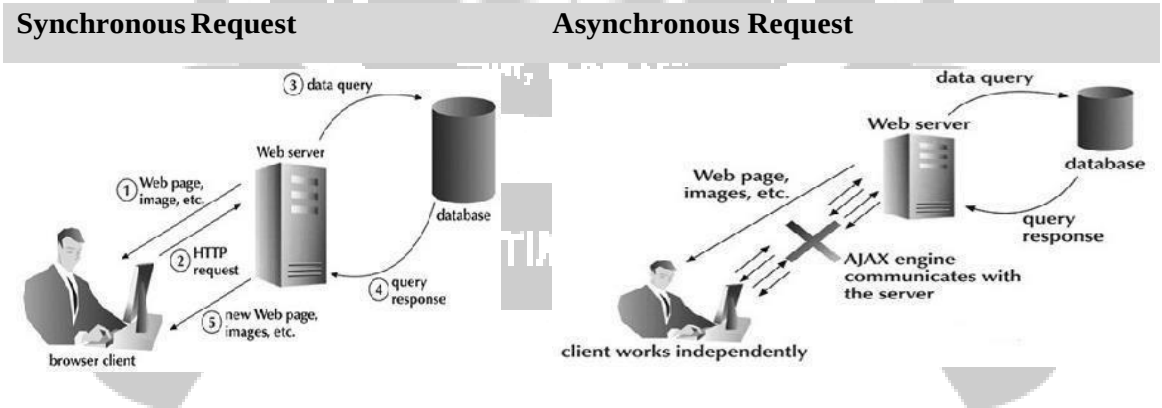


Fig 5.1: Working of AJAX



5.1 CLIENT SERVER ARCHITECTURE

The Ajax provides a rich and diverse set of product that range from hundreds of suppliers. This diversity provides IT managers and Web developers with the ability to choose the optimal architectural approach and best products among multiple vendors.

Most Ajax technologies transform a platform-independent definition of the application into the appropriate HTML and JavaScript content that is then processed by the browser to deliver a rich user experience. Some Ajax designs perform most of their transformations on the client. Others perform transformations on the server.

Client side vs server side transformations

- **Client-side Ajax transformations**
  - With client-side Ajax, the Ajax engine runs on the client.
  - The server delivers Web content (HTML, CSS, JavaScript, etc.) which is processed by the client-side Ajax engine into revised Web content.

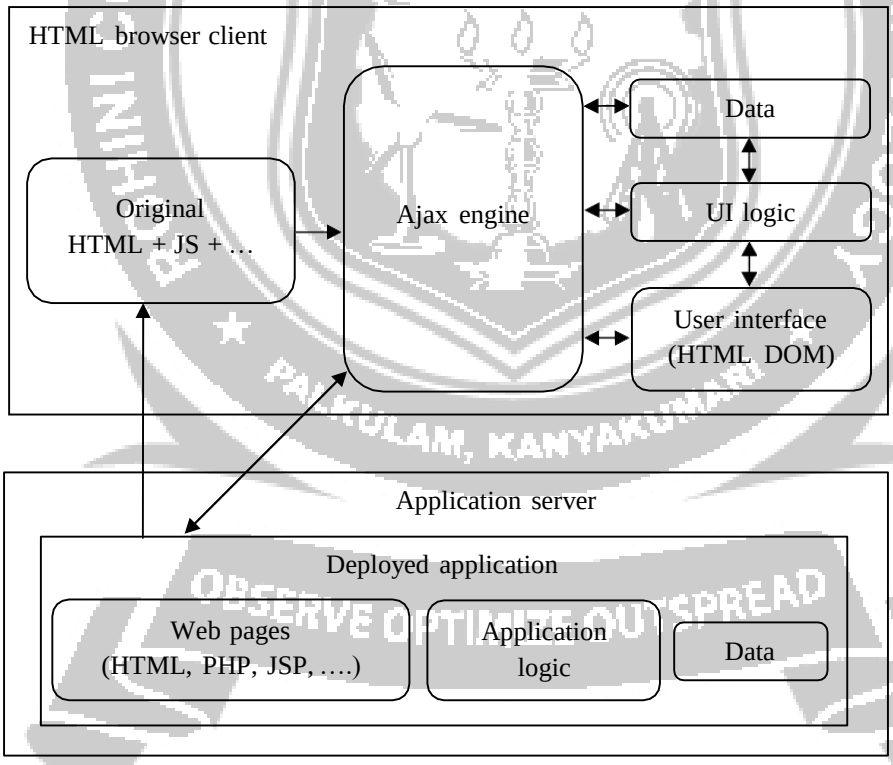


Fig 5.2 AJAX Client transformations

- The browser renders the revised HTML/etc. content that comes out of the Ajax engine.

- With this architecture, the application development team typically provides the following server-side components: Web pages (e.g., *.html, *.php, *.jsp, *.asp), application logic (e.g., Java) and data management (e.g., via a SQL database and/or Web Services)
- The client-side component includes client-side user interface logic, such as event handlers.
- The advantage of this option is the independence from the server side technology.
- The server code creates and serves the page and responds to the client's asynchronous requests.

This way either side can be swapped with another implementation approach**.**

➢ **Server-side transformations**

- For server-side Ajax, an Ajax server component performs most or all of the Ajax transformations.

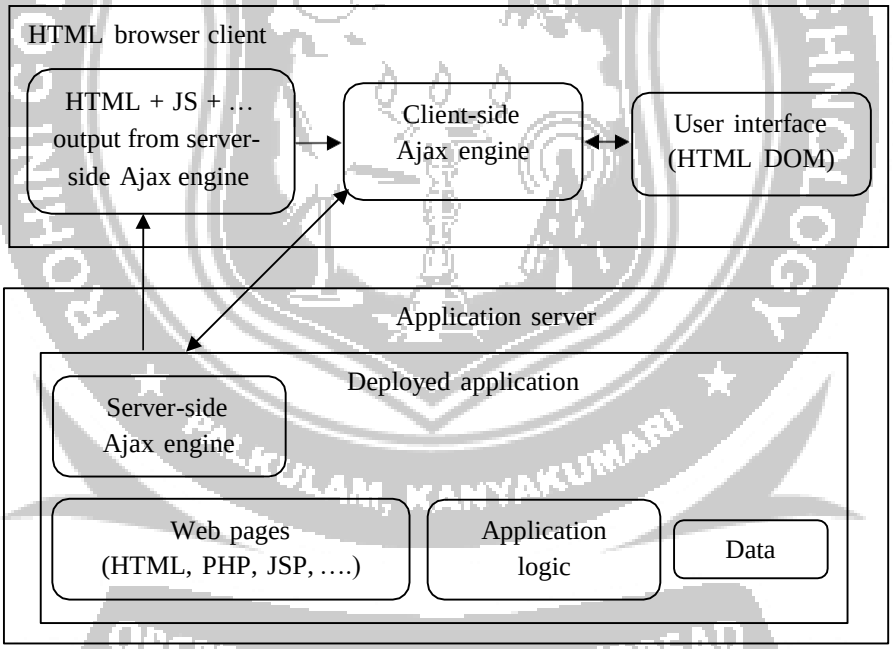


Fig 5.3 AJAX Server transformations

- The server component generates the necessary Web content (HTML, CSS, JavaScript, etc.) to deliver the desired user experience.
- The server-side Ajax toolkit downloads its own client-side Ajax library which communicates directly with the toolkit's server-side Ajax component.

- With this architecture, the application development team typically only provides server-side components (Web pages, application logic, and data management) and entrusts client-side logic to the Ajax toolkit.
- The main benefit of this approach is that it allows the use of server-side languages for debugging, editing, and refactoring tools with which developers are already familiar
- The disadvantage of this approach is the dependence on a particular server-side technology.
- As a general rule, server-side Ajax frameworks expect application code to be written in the server-side language.
- These frameworks typically hide all the JavaScript that runs in the browser inside widgets, including their events.

Single DOM vs Dual DOM

Single DOM

Some Ajax runtime toolkits use a Single-DOM approach where the toolkit uses the browser's DOM for both any original source markup and any HTML+JavaScript that results from the toolkit's Ajax-to-HTML transformation logic. This is same as Fig 5.2 In the above example, the developer is using tree widgets from an Ajax runtime library. The original HTML markup (un-shaded) and the additional HTML markup inserted by the Ajax toolkit (shaded) is given in the left. The DOM objects that correspond to the elements in the HTML markup (e.g., the DOM object that represents a particular element), where JavaScript/DOM objects from unshaded objects correspond to original HTML markup and shaded objects are ones that have been inserted by the Ajax toolkit is given in the right.

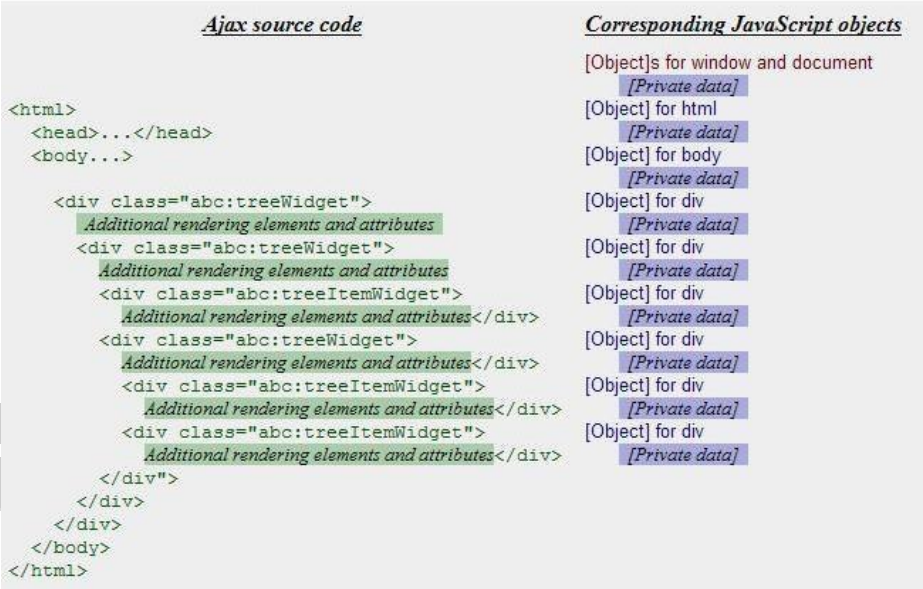


Fig 5.4 Manipulation of Single DOM

Typically, the Ajax toolkit inserts various rendering constructs such as <img> , , and <p> elements, inline within the original HTML markup (e.g., adding child elements to an existing element), thereby providing the various graphics and text necessary to produce the desired visual representation for the tree widgets. The shaded sections on the right reflect the private data that Ajax libraries typically add to various DOM and JavaScript objects in order to store private data, such as run-time state information. The Single-DOM approach is particularly well-suited for situations where the developer is adding Ajax capability within non-Ajax DHTML application.

Dual DOM

Other Ajax runtime toolkits adopt a Dual-DOM approach that separates the data for the Ajax-related markup into an Ajax DOM tree that is kept separate from the original Browser DOM tree. The Dual-DOM approach has two types:

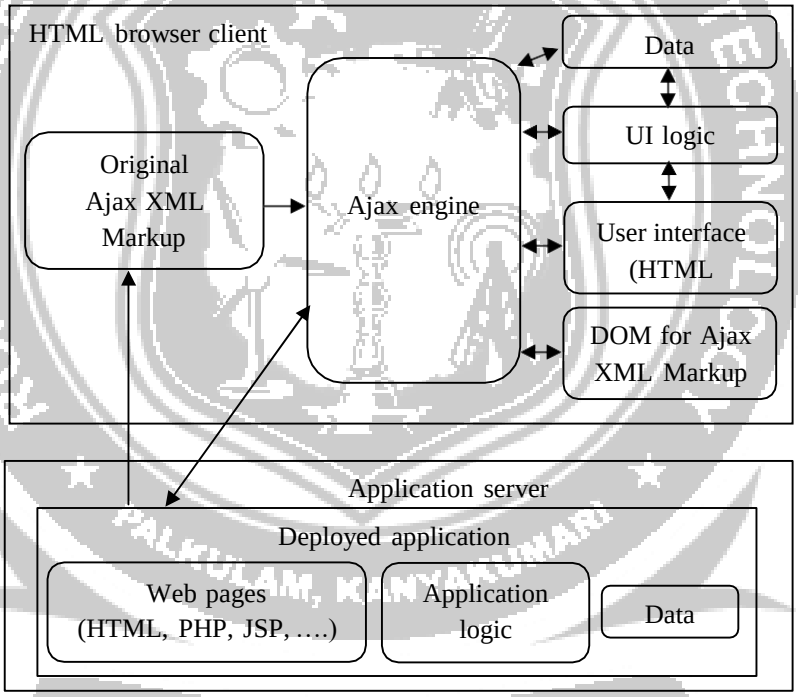


Fig 5.5 Dual DOM

With **Client-side Dual-DOM**, the second DOM tree typically consists of a separate tree of JavaScript objects. With **Server-side Dual-DOM,** the second DOM tree is stored on the server.

Fig 5.6 Manipulation of Dual DOM

There are two DOMs : the **HTML DOM and the XML DOM** corresponding to the Ajax-specific XML markup for the user interface elements.

The above example shows a separate file, "myapp.abc", which contains the user interface definition for the tree widgets, which in this case are to be placed into the HTML tree inside the element with id="abctarget". Even though the example shows the use of a separate file, some Dual-DOM Ajax runtime libraries support inline XML. In either case, a Dual-DOM Ajax runtime library builds a separate DOM tree, typically using its own XML parser rather than relying on the browser's HTML parser.

Sometimes the separate DOM tree is attached to the 'window' or 'document' objects. With this approach, in model view controller (MVC) terms, the Ajax DOM can be thought of as the model, the Browser DOM as the generated view, and the Ajax runtime toolkit as the controller. It is usually necessary to establish bidirectional event listeners between the Ajax DOM and the Browser DOM in order to maintain synchronization. Sometimes having a separate Ajax DOM enables a more complete set of XML and DOM support, such as full support for XML Namespaces, than is possible in the Browser DOM.

Dual-DOM (server-side)

Some Ajax technologies combine server-side Ajax transformations with a Dual-DOM approach. The key difference between Server-side Dual-DOM and Client-side Dual-DOM is that, with Server-side Dual-DOM, the Ajax DOM and most user interface logic is managed on the server. In this scenario, the primary job of the client Ajax engine is to reflect back to the server any interaction state changes, deferring data handling, UI state management and UI update logic to the server. Server-side Dual-DOM enables tight application integration with server-side development technologies such as Java Server Faces (JSF).

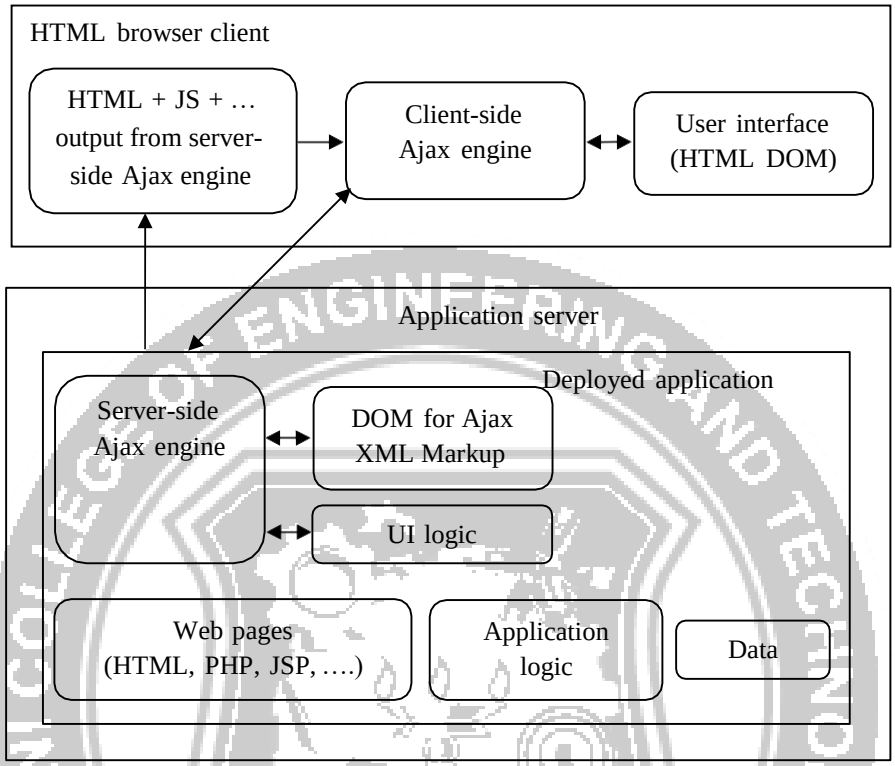


Fig 5.7 Server side Dual DOM

5.2XML Http Request Object and CALLBACK()

The XMLHttpRequest object is used to exchange data with a server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page. **XMLHttpRequest (XHR)** is an API that can be used by JavaScript, JScript, VBScript, and other web browser scripting languages to transfer and manipulate XML data to and from a webserver using HTTP, establishing an independent connection channel between a webpage's Client-Side and Server-Side. The data returned from XMLHttpRequest calls will often be provided by back-end databases. Besides XML, XMLHttpRequest can be used to fetch data in other formats, e.g. JSON or even plain text.

Creatingan XMLHttpRequest Object

Syntax:

variable=new XMLHttpRequest(); (new version)

variable=new ActiveXObject("Microsoft.XMLHTTP"); (old version)

Processing Requests in AJAX

The following are the sequence of operations request is initiated:

- A client event occurs.
- An XMLHttpRequest object is created.
- The XMLHttpRequest object is configured.
- The XMLHttpRequest object makes an asynchronous request to the Webserver.
- The Webserver returns the result containing XML document.
- The XMLHttpRequest object calls the callback() function and processes the result.
- The HTML DOM is updated.

- **A client event occurs:**

- A JavaScript function is called as the result of an event.
  - **Example:** validateUserId() JavaScript function is mapped as an event handler to an onkeyup event on input form field whose id is set to "userid".

```
<input type="text" size="20" id="userid" name="id"onkeyup="validateUserId();">.
```

- **XMLHttpRequest object is created**

```
varajaxRequest; // AJAX variable
functionajaxFunction()
{ try
{ // This code is for browsers Opera 8.0+, Firefox, Safari
ajaxRequest =new XMLHttpRequest();        }
catch (e)
   {    // Internet Explorer Browsers
Try         {
ajaxRequest = new ActiveXObject("Msxml2.XMLHTTP");        }
catch (e)          {
try{
ajaxRequest = new ActiveXObject("Microsoft.XMLHTTP");                }
catch (e){
              // Something went wrong
alert("Your browser broke");
return false;                    }       }       } }
```

- **The XMLHttpRequest object is configured**

```
functionvalidateUserId()
{ ajaxFunction(); // Here processRequest() is the callback function.
ajaxRequest.onreadystatechange = processRequest;
if (!target) target = document.getElementById("userid");
varurl = "validate?id=" + escape(target.value);
ajaxRequest.open("GET", url, true);
ajaxRequest.send(null);       }
```

- **Making an asynchronous request to the Webserver**

  This is done using the XMLHttpRequest object ajaxRequest. Assume that the user enters Sona in the userid box, then in the above request, the URL is set to "validate?id=Sona".

- **Webserver Returns the Result Containing XML Document**

  Server-side script is implemented as follows:

  - Get a request from the client.
  - Parse the input from the client.
  - Do required processing.
  - Send the output to the client.

If we assume that the user is going to write a servlet, then:

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws
IOException, ServletException
{   String targetId = request.getParameter("id");
if ((targetId != null) && !accounts.containsKey(targetId.trim()))
  { response.setContentType("text/xml");
response.setHeader("Cache-Control", "no-cache");
response.getWriter().write("true");  }
else   {
response.setContentType("text/xml");
response.setHeader("Cache-Control", "no-cache");
response.getWriter().write("false");  } }
```

- **Callback Function processRequest() is Called**

The callback function is responsible for checking the progress of requests, identifying the result of the request and handling data returned from the server. Callback functions also serve as delegators, handing off to other areas of the application code. The XMLHttpRequest object was configured to call the processRequest() function when there is a state change to the readyState of the XMLHttpRequest object. Now this function will receive the result from the server and will do the required processing. As in the following example, it sets a variable message on true or false based on the returned value from the Webserver.

```
functionprocessRequest()
{ if (req.readyState == 4)
    { if (req.status == 200)
        { var message = ...;
            ...
    }
```

- **The HTML DOM is updated.**

This is the final step and in this step, the HTML page will be updated. It happens in the following way:

- JavaScript gets a reference to any element in a page using DOM API.
- The recommended way to gain a reference to an element is to call.

  document.getElementById("userIdMessage"), // userIdMessage is ID attribute

  // of an element appearing in the HTML document
- JavaScript may now be used to modify the element's attributes; modify the element's style properties; or add, remove, or modify the child elements.

```
<script type="text/javascript">
<!— functionsetMessageUsingDOM(message)
{ varuserMessageElement = document.getElementById("userIdMessage");
varmessageText;
if (message == "false")
{ userMessageElement.style.color = "red";
messageText = "Invalid User Id"; }
else {
```

```
userMessageElement.style.color = "green";
messageText = "Valid User Id";    }
varmessageBody = document.createTextNode(messageText);
if (userMessageElement.childNodes[0])
{  userMessageElement.replaceChild(messageBody,  userMessageElement.childNodes[0]);
}
 Else   { userMessageElement.appendChild(messageBody);   } }
</script> <body> <div id="userIdMessage"><div> </body>
```

XMLHttpRequest Methods

Method	Description
abort()	Cancels the current request.
getAllResponseHeaders()	Returns the complete set of HTTP headers as a string.
getResponseHeader(headerName)	Returns the value of the specified HTTP header
open(method, URL) open(method, URL, async) open(method, URL, async, userName) open(method, URL, async, userName, password)	Specifies the method, URL, and other optional attributes of a request. The method parameter can have a value of "GET", "POST", or "HEAD". Other HTTP methods, such as "PUT" and "DELETE" may be possible. The "async" parameter specifies whether the request should be handled asynchronously or not. "true" means that the script processing carries on after the send() method without waiting for a response, and "false" means that the script waits for a response before continuing script processing.
send(content)	Sends the request.
setRequestHeader(label, value)	Adds a label/value pair to the HTTP header to be sent.

XMLHttpRequest Properties

- **onreadystatechange:** An event handler for an event that fires at every state change.
- **readyState:** The readyState property defines the current state of the XMLHttpRequest object.

State	Description
0	The request is not initialized.
1	The request has been set up.
2	The request has been sent.
3	The request is in process.
4	The request is completed.

- readyState = 0 After the user have created the XMLHttpRequest object, but before the call of the open() method.
- readyState = 1 After the user have called the open() method, but before the call of send().
- readyState = 2 After the user have called send().
- readyState = 3 After the browser has established a communication with the server, but before the server has completed the response.
- readyState = 4 After the request has been completed, and the response data has been completely received from the server.
  - responseText:Returns the response as a string.
  - responseXML:Returns the response as XML. This property returns an XML document object, which can be examined and parsed using the W3C DOM node tree methods and properties.
  - Status:Returns the status as a number (e.g., 404 for "Not Found" and 200 for "OK").
  - statusText:Returns the status as a string (e.g., "Not Found" or "OK").