|  |
| --- |
| [[https://myetudes.org/etudes-melete-tool/images/printer.png](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385328) Send to Printer](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385328) | [Close Window](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385328) |
|  | |
| 17. Ajax  17.1. What is Ajax?  *Copyright (c) 2014, Rula Khayrallah*  Ajax stands for Asnychronous JavasScript and XML.  It refers to a set of techniques for **exchanging data with the server without causing pages to reload**.  It allows web pages to be updated **asynchronously** by exchanging **small amounts of data** with the server. This of course results in a more responsive web application and a better user experience.  Google Maps is one of the best-known examples of Ajax applications.  AJAX is based on:  JavaScript  HTML and CSS  he Document Object Model  The XMLHttpRequest API to exchange data asynchronously with a server. Even though it is possible to implement Ajax by using < iframe > or < script > tags to communicate with the server,   modern browsers all support the XMLHttpRequest API and the recommended way to implement Ajax is using the XMLHttpRequest object.  A data exchange format:   even though the x in Ajax stands for XML and XMLHttpRequest  includes XML in its name, the use of XML is optional and becoming rare.  The techniques support any text based format and JSON is currently used more often.  17.2. Sending Requests to the Server  *Copyright (c) 2014, Rula Khayrallah*  **The XMLHttpRequest object** is used to exchange data with the server.  We have seen last week (module 15) that clients and servers communicate using the HTTP protocol.  They exchange HTTP requests and HTTP responses.  From the client side, to initiate the communication, **we first create an XMLHttpRequest object:**  **var request = new XMLHttpRequest();**  Here the variable request refers to the newly created object.  Then we specify the different parts of the HTTP request (method, url, headers, body) and send them.  Let’s see how to do that next.  **The open() method:**  We can use the open() method on our newly created request object to set the HTTP request method and url.  The general syntax for the open() method is as follows:  open (method, url, async, user, password)  The method parameter specifies**the HTTP request method or verb to use.  The two most common methods are GET and POST.**  **GET** is used in most requests to **request** data from the server.  **POST**is typically used to **submit** data to the server.  We also use POST when we don’t want a cached version of the server’s response and when we are sending sensitive information.  The url parameter specifies**the url that is requested.**It points to a 'resource' on the server.  This can be any kind of 'resource' such as a text file (txt or xml) or a program (asp or php).  The url specified here is usually a relative url (relative to the URL of the current document).  If we specify an absolute url, it must include the same protocol, hostname, and port number as the current document.  Otherwise it will usually result in an error, because it violates the same-origin policy.  Note that the url may also include a query string that allows the client to pass additional parameters to the server.  **async is an optional parameter that specifies whether or not the request is sent asynchronously.   If we do not specify it, it defaults to true (asynchronous).**  The user and password parameters are optional and may be used for authentication purposes.  So for a GET request, to get the "data.txt” file from the server, we write:  var request = new XMLHttpRequest();  request.open('GET', 'data.txt');  To send the request to the server, we use the send() method.  The send method may include the request body, if present.  GET requests don’t include a body, so for the GET request above, we simply write:  request.send();  So putting it all together, to send a GET request for the file 'data.txt', we have:  function sendRequest() {      var request = new XMLHttpRequest();      request.open('GET', 'data.txt');      request.send();  }  Note that the open method and the send method are invoked on the newly created XMLHttpRequest object.  Let’s take a look at how a POST request would be implemented.  Let’s assume that we would like to send some JSON encoded data to the server.  We first create the XMLHttpRequest object:  var request = new XMLHttpRequest();  Then we set the method and the resource.  Note that the resource name, someurl, will depend on the specific setup of your server.  request.open("POST", "someurl");  We also have to set the request header.  We can do that with the setRequestHeader method.  We are basically telling the server that the request body contains JSON encoded data.  request.setRequestHeader("Content-Type", "application/json");  And finally we send our request body to the server with send.  The variable data contains the data that we need to send to the server:  request.send(JSON.stringify(data));  So putting it all together, to send a POST request for some JavaScript object data, we have:  function postRequest(){      var request = new XMLHttpRequest();      request.open("POST", "someurl");  // someurl depends on your server      request.setRequestHeader("Content-Type", "application/json");      request.send(JSON.stringify(data));  }  Note that here again, the methods open, setRequestHeader and send are all invoked on the newly created XMLHttpRequest object.  17.3. Handling Asynchronous Responses  *Copyright (c) 2014, Rula Khayrallah*  **When we send our request asynchronously, our program does not block until the server responds:**  that’s a good thing, it can go ahead and perform other tasks and it can be more responsive to the user.  However **the program needs to be somehow notified when the request is complete** so that it can retrieve it.  To do that we can listen for **'readystatechange'**events on the XMLHttpRequest object.  The 'readystatechange' event is based on the readyState property of the XMLHttpRequest object. It is triggered when the readyState property changes.  The readyState property describes the status of the request.  It has one of the following values:  0:  UNSENT - open() has not been called yet  1: OPENED  2: HEADERS\_RECEIVED  3: LOADING – loading the response  4: DONE  **To listen for readystatechange events, we set the onreadystatechange property of the XMLHttpRequest object to our event handler function.**  We could also use addEventListener() but the onreadystatechange property is commonly used here.  To set the onreadystatechange property using an anonymous function expression, we write:  request.onreadystatechange = function() {   if (request.readyState === 4 && request.status === 200){       // add code here to retrieve the response      }   } ;  The anonymous function above will now be called whenever a readystatechange event occurs on our request object.  Note that because **the readystatechange event is triggered several times before the response is complete, we first need to check that the request is complete when handling this event**:  if (request.readyState === 4...  And because the request may or may not succeed, we must also check the HTTP status code.  As we have seen before, 200 means OK.  So we have:  if (request.readyState === 4 && request.status === 200) {   …  We'll see next how to retrieve the response.  17.4. Retrieving the Responses  *Copyright (c) 2014, Rula Khayrallah*  A complete HTTP response includes **a status code, response headers, and a response body**. We have access to all of these through the properties and methods of the XMLHttpRequest object.  We have already seen how to check the status of the response with request.status.  request.statustext is also available:  it is a string with values such as "OK" or "Not Found".  We can use getResponseHeader() to access the response headers.  To get the "Content-Type" information from the response header, we write:  var type = request.getResponseHeader("Content-Type");  We can use the responseText property to access the response body as plain text.  The responseXML property allows us to access the response body when it is an XML or XHTML document.  To display the response body on our web page in an element with id response, we can write:  document.getElementById("response").textContent = request.responseText;  17.5. Simple Demo  *Copyright (c) 2014, Rula Khayrallah*  To illustrate how the various steps involved in communicating with the server come together, we’ll use a basic example.  **We'll use the following folder hierarchy to organize the files:**  Our server program - shown in purple - will go in the NodeServer folder.  The NodeServer folder includes two subfolders:  html and scripts.  The client side JavaScript programs - shown in green - will go in the scripts folder.  The html, css and images - shown in pink - will go in the html folder.  **Let’s start with the HTML source file ajaxdemo.html**.  **ajaxdemo.html**  <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8">  <title>JavaScript for Programmers</title>  </head>  <body>  <h2> Ajax Demo</h2>  <p>Click on the button to send an asynchronous request to the server </p>  <input id = "request" type="button" value="CLICK">  <p id="response"></p>  <script defer src="../scripts/ajaxdemo.js"></script>  </body>  </html>  It’s a very simple web page.  You can open it and display it in your browser to see what it looks like but the Ajax features will not be active: **the XMLHttpRequest API does not work with the FILE: scheme so we’ll need to setup a server.**  The goal is to have our client-side script ajaxdemo.js respond to a user click by sending an asynchronous request to the server and then displaying the additional data received in the <p> element with id = "response".  **Let’s write our client-side script ajaxdemo.js** to handle that.  We'll save our client-side JavaScript file ajaxdemo.js in the scripts subfolder.  **ajaxdemo.js**  function sendRequest() {      // create an XMLHttpRequest object      var request = new XMLHttpRequest();      // Specify a GET request for the file data.html      request.open("GET", "../html/data.html");      request.send();      // define the function to be called when the response is received.      request.onreadystatechange = function () {         // check that the response is complete and the request was successful         if (request.readyState === 4 && request.status === 200){            // Display the response            document.getElementById("response").innerHTML = request.responseText;         }      };  }  // Add an event listener to call sendRequest when the button is clicked  document.getElementById("request").addEventListener("click", sendRequest);  We also need to**create the data.html** file that will be requested by the client.  For our demonstration purposes, we’ll create it with just one line:  **data.html**  <span>This is the additional data requested from the server.</span>  We'll save the file data.html in the html subfolder.  Now we are ready  to **setup a server**to serve ajaxdemo.html and the corresponding ajaxdemo.js as well as data.html.  We'll use the following Node server that is very similar to the  server you wrote in a previous assignment.  In addition to html pages, our new server can serve client-side JavaScript code, css files, json files and so on.  **server.js**  // Simple node server - CS22A    // The following function will be called whenever  // the server receives a request.  function servePage(request, response) {        // MIME types supported by this server      var supportedTypes = {          '.html': 'text/html; charset = UTF-8',          '.txt': 'text/plain; charset = UTF-8',          '.js': 'application/javascript; charset = UTF-8',          '.appcache': 'text/cache-manifest; charset = UTF-8',          '.css': 'text/css; charset = UTF-8',          '.json': 'application/json; charset = UTF-8',          '.gif': 'image/gif'      }        // Extract the filename and extension from the request.      var filename = '.'+ url.parse(request.url).pathname;        // If the user does not enter a file name,      // we serve the page home.html    if (filename === './') {      filename = './html/home.html';    }      // Get the extension of the requested resource so we can determine the type      var extension = path.extname(filename)      var type = supportedTypes[extension];  // type implied by extension        // Read the file asynchronously      fs.readFile( filename, function( err, content) {          if (err) { // If there is an error, set the status code              response.writeHead( 404,                                 {'Content-Type': 'text/plain; charset = UTF-8'});              response.write( err.message); // Include the error message body              response.write( ' - The page requested is not found.');              response.end(); // Done          } else { // Otherwise, the file was read successfully.              response.writeHead( 200, // Set the status code                                 {'Content-Type': type});              response.write(content); // Send file contents as response body              response.end();          }      });    };    // load the url module  var url = require('url');  var path = require('path');  // Load the file system module  // Load the file system module  var fs = require("fs");  // load the http module  var http = require('http');    // create a server object  var server = http.createServer(servePage);  server.listen(8080);  console.log('Server running at http://localhost:8080');  **From the Node command line (or terminal window on a Mac),  we start the server by navigating to the folder that contains it and then typing:**  node server.js  Then we start Firefox and type [http://localhost:8080/html/ajaxdemo.html](http://localhost:8080/ajaxdemo.html) in the address bar.  The following should be displayed:    Open the Firebug Console and make sure you select Show XMLHttpRequests from the console drop down menu as shown below:    Then go to the Ajax Demo web page and click on the button.  You get the following:  Note that the additional text displayed under the button came from the server’s response.  Note also that you get details about the request and response in the Firebug console window.  Click on the + sign next to the GET to see details about the request headers and the response.    17.6. Ajax with JSON  *Copyright (c) 2014, Rula Khayrallah*  To illustrate Ajax calls with JSON, we'll go back to our business review application.  We'll assume that we have initially displayed a list of businesses (alphabetically sorted by name), along with their rating, price and distance.  We would like to give the user the option to request some additional data about a particular business.  Our goal is to get that data and display it to the user **without a page reload**.  We'll assume that the data is stored on the server in a JSON encoded file with a name based on the business name.  So initially, our web page looks like:  And then, when the user requests more information about 'California Foods',  the client side JavaScript code gets the additional information through an Ajax call to the server, and the following is displayed:  The HTML source document, review.html is available under Resources.  Here's the style sheet that we'll use.  review.css  body {              background-color:#fdf5e6;              font-family: Sans-Serif;              text-align:center;  }  table {              margin-left:auto;              margin-right:auto;  }  #business{              border-collapse:collapse;  }  td {              border:1px solid #eed9ba;  }  th,  input{              background-color:#eecbad;              color:#ffffff;  }    Our client side JavaScript code:    reviewajax.js  /\*   \* CS 22A -Business Review  - Ajax Implementation   \*/  (function () {      'use strict';      function getAjaxData(event) {          // create an XMLHttpRequest object - the variable request now points to that object          var request = new XMLHttpRequest();          // the requested filename is given by the input element id          // the json files are kept in a separate data folder          var filename = '../data/' + event.target.id + '.json';          // We invoke the methods (open and send) on the XMLHttpRequest object          // denoted by the variable request.          // Specify a GET request for the JSON file          request.open('GET', filename);          request.send();          // define the function to be called when the response is received.          // we specify the onreadystatechange property of the XMLHttpRequest object          // denoted by the variable request.          request.onreadystatechange = function () {              // check that the response is complete and the request was successful              if (request.readyState === 4 && request.status === 200) {                  // Display the response - responseText is the JSON encoded string                  displayInfo(request.responseText);              }          };      }      function displayInfo(jsonString) {          // convert the JSON encoded string to object          var jsonObj = JSON.parse(jsonString);          // build a string from the properties and their values.          var info = '';          for (var prop in jsonObj) {              info += '<p>' + prop + ': ' + jsonObj[prop] + '<p>';          }          // display the additional info in the description html element            document.getElementById('description') .innerHTML = info;      }      // Register event handlers to get the additional info if requested      document.getElementById('business') .addEventListener('click', getAjaxData, false);  }());    Note that to minimize namespace pollution, we have wrapped our code in an anonymous function as described in module 13.3  Finally we'll need our JSON encoded data files.  We'll save our JSON files in a separate 'data' subfolder.    Let's create two files, for demonstration purposes:  **californiafoods.json** (for California Foods):  {"Name":"California Foods",  "Address":"123 El Monte Road, Los Altos Hills, CA 94022",  "Phone":"650-555-1234",  "Website":"cafoods.com",  "Most Helpful Review":"Good seafood and fresh salads"}    And **everythingpizza.json** (for Everything Pizza):  {"Name":"Everything Pizza",  "Address":"123 Castro St, Mountain View, CA 94041",  "Phone":"650-555-5678",  "Website":"everythingpizza.com",  "Most Helpful Review":"The thin crust vegetarian pizza is the best."}    Now we are ready to start our server (server.js from the previous section) from the Node command prompt:  node server.js  Then we point our browser to: localhost:8080/html/review.html.  The initial web page is shown. When we click on 'More Info', we initiate the Ajax call and get the additional information on 'California Foods' or 'Everything Pizza'.  If we open Firebug and make sure 'Show XMLHttpRequests' is selected under the Console tab, we can see the details of the Ajax call under the JSON and the Response tabs:        17.7. Ajax with jQuery  *Copyright (c) 2014, Rula Khayrallah*  jQuery includes some methods and functions that make Ajax simpler to implement. We'll demonstrate the use of **load** and **getJSON** in the following examples.  Note that jQuery provides two additional functions **jQuery.get()** and **jQuery.post()**that allow us to send asynchronous GET and POST requests to the server without having to deal with the details of the XMLHttpRequest object. Finally, the**jQuery.ajax()** function allows us to specify a more generic request with more control over the various request options.  More details about these functions are available at <http://api.jquery.com/>.  **The load() method:**  To follow along, modify the html file used in the previous demo ajaxdemo.html to include jQuery:  ajaxdemo.html  <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8">  <title>JavaScript for Programmers</title>  </head>  <body>  <h2> Ajax Demo</h2>  <p>Click on the button to send an asynchronous request to the server </p>  <input id = "request" type="button" value="CLICK">  <p id="response"></p>  **<script defer src="../scripts/jquery-1.11.1.js"></script>**  <script defer src="../scripts/ajaxdemo.js"></script>  </body>  </html>  We now have to make sure that the file jquery-1.11.1.js is in the scripts subfolder.  The **load() method** **may be used to load a given url into each of the elements specified by a jQuery selector.**  In our previous demo, we could have rewritten our client side script ajaxdemo.js as follows:  ajaxdemo.js  function sendRequest(){      // Load additional data from the server into the response element  **$("#response").load("../html/data.html");**  }  // Add an event listener to call sendRequest when the button is clicked  $("#request").on("click", sendRequest);  We can even select parts of the loaded document by specifying a jQuery selector following the url.  Let’s modify our data.html file to include the following two lines with ids line1 and line2.  data.html  <span id = "line1" >This is the first line of the additional data requested from the server.</span>  <span id = "line2" >This is the second line of the additional data requested from the server.</span>  Then to load only line2 in the response element of ajaxdemo.html, we can modify ajaxdemo.js as follows:  function sendRequest(){      // Load additional data from the server into the response element      $("#response").load("../html/data.html **#line2**");  }  // Add an event listener to call sendRequest when the button is clicked  $("#request").on("click", sendRequest);    The next step is to run the server from the node command window:  node server.js  Then enter http://localhost:8080/html/ajaxdemo.html in the address bar in Firefox.  The initial Ajax demo web page is displayed and after we click the button we get:  **The getJSON() function:**  The getJSON jQuery function also allows us to seamlessly load JSON  encoded data directly from the server without having to go into the details of the HTTP requests and responses.  Let's revisit our business review Ajax implementation and modify our JavaScript code to use jQuery.  In addition to getJSON, we'll use the jQuery $.each function that allows to iterate over any object. The callback function is passed a property name and value each time.   Note that in this simple example, we are not handling the error case where the JSON file does not exist.  reviewajaxjq.js  /\*   \* CS 22A -Business Review  - Ajax Implementation   \*/  (function () {      'use strict';      function getAjaxData(event) {          var filename = '../data/' + event.target.id + '.json';          $.getJSON(filename, function (jsonObj) {              var info = '';              $.each(jsonObj, function (prop, value) {                  info += ('<p>' + prop + ': ' + value + '</p>');              });              $('#description') .html(info);          });      }      // Register event handlers to get the additional info if requested      $('input') .on('click', getAjaxData);  }());  Note that we'll also have to modify our source html document review.html to include jquery.  We can do that by adding the following line right before the existing <script>  tag.  **<script defer src="../scripts/jquery-1.11.1.js"></script>**  We'll also have to make sure we point to the new jquery implementation of the JavaScript code, so we modify the existing <script> tag as follows:  <script defer src=**"../scripts/reviewajaxjq.js">**</script>    We can save our modified html document as reviewjq.html.  Now we can test the jQuery code by running the server from the Node command line, then entering http://localhost:8080/html/reviewjq.html in the address bar in Firefox.  If we open Firebug, we can verify the details of the Ajax calls that take place when we click on the More Info buttons.  17.8. Limitations  *Copyright (c) 2014, Rula Khayrallah*  There are a few issues surrounding Ajax applications that we need to be aware of:  **Security:**  Care must be taken not to transfer sensitive data as plain text in Ajax calls.  Some consider that the increased communication between the client and the server may  make the application more vulnerable to security threats.  **Search Engine Indexing:**  The content of AJAX applications will not usually appear in search engine results.  Web crawlers do not crawl JavaScript code, so unless we use some other technique to make our content available to search engines, it will not appear in the search engine results.  **Back Button and Bookmarking:**  History management and bookmarking are more challenging to implement in Ajax applications.  We'll see how to handle that from within our JavaScript program in the next module. |  |