Net-Centric Computing

Javascript: Ajax II

Dr. Simon Fan, Associate Professor of CSSE

xfan@psu.edu

Objectives

- JSONP: HTTP by <script>
- Uploading files;
- HTTP progress events
- Responses

(1)Use a hidden <**iframe**> as Ajax transport

- The <iframe> need not be visible to the user; it can be **hidden** with CSS.

- The process:
  1) The script first encodes query information into a **URL** (subject to same-origin policy) and then sets the **src** property of the <**iframe**> to that URL.
  2) The request goes to the server
  3) The server creates an HTML document containing its response and sends it back to the web browser.
  4) The web browser displays it in the <**iframe**>.
  5) The script can then access the server's response by **traversing the document object of the <iframe>**.

(2) **JSONP**: Use a dynamic <**script**> as Ajax transport

- The <**script**> element has a **src** property that can be set to initiate an HTTP GET request.
- This Ajax transport is known as **JSONP**, because it is a protocol using the JSON data format
  - The server's response takes the form of **JSON-encoded data** that is automatically "decoded" when the script is executed by the JavaScript interpreter in a browser.
- Doing HTTP scripting with <script> elements is particularly attractive because they are **not subject to the same-origin policy** and can be used for cross-domain communication.

JSONP: HTTP by <script>

Note: XMLHttpRequest API is NOT explicitly used

- Set the **src** attribute of a <script> (and insert it into the document if it isn't already there), the browser will
  - generate an HTTP request to download the URL you specify, and
  - automatically decode (i.e., execute) response bodies that consist of JSON-encoded data
- Plain JSON response: a JSON object `[1, 2, {"buckle": "my shoe"}]`
- JSONP response: Plain JSON response surrounded with parentheses and prefixed with the name of a JavaScript function.

```
handleResponse(
[1, 2, {"buckle": "my shoe"}]
)
```

- A client can tell the server what function name is to be used as padding in the response
  - This can be done by adding another query parameter, say "jsonp" (or "callback" in Node.js), with a function name being its value

JSONP example

```
// Make a JSONP request to the specified URL and pass the parsed response
// data to the specified handler. Add a query parameter named "callback" to
// the URL to specify the name of the callback function for the request.
function getJSONP(url, handler) {
    // Create a unique callback name just for this request
    var cbnum = "cb" + getJSONP.counter++; // Increment counter each time
    var cbname = "getJSONP." + cbnum;      // As a property of this function
    // Add the callback name to the url query string using form-encoding
    // We use the parameter name "callback", which is required by Node.js.
    if (url.indexOf("?") === -1)   // URL doesn't already have a query section
        url += "?callback=" + cbname; // add parameter as the query section
    else                           // Otherwise,
        url += "&callback=" + cbname; // add it as a new parameter.
    // Create the script element that will send this request
    var script = document.createElement("script");
    //script.type="application/javascript";

    // Define the callback function that will be invoked by the script
    getJSONP[cbnum] = function(response) {
        try {
            handler(response); // Handle the response data
        }
        finally {              // If handler threw an error
            delete getJSONP[cbnum];      // Delete this callback function
            script.parentNode.removeChild(script); // Remove script
        }
    };
    // Now trigger the HTTP request
    script.src = url;                    // Set script url
    document.body.appendChild(script); // Add it to the document
}
getJSONP.counter = 0;  // A counter we use to create unique callback names
```

HTTP Progress Events

- The XHR2 draft specification (implemented in Firefox, Chrome, and Safari) defines events for monitoring the **download** of an HTTP response.

Event name	Interface	Dispatched when…
readystatechange	`Event`	The `readyState` attribute changes value, except when it changes to UNSENT.
loadstart	`ProgressEvent`	The fetch initiates.
progress	`ProgressEvent`	Transmitting data.
abort	`ProgressEvent`	When the fetch has been aborted. For instance, by invoking the `abort()` method.
error	`ProgressEvent`	The fetch failed.
load	`ProgressEvent`	The fetch succeeded.
timeout	`ProgressEvent`	The author specified timeout has passed before the fetch completed.
loadend	`ProgressEvent`	The fetch completed (success or failure).

XMLHttpRequest: retrieving response

- The XMLHttpRequest object is usually used **asynchronously**: the send() method **returns immediately** after sending the request.
  - Client-side JavaScript is single-threaded and when the send() method blocks, it typically freezes the entire browser UI.
  - Similarly, if the server you are connecting to is responding slowly, your user's browser will freeze up.
- To be notified when the response is ready, code must listen for the readystatechange events on the XMLHttpRequest object.
- The **readyState** property is an integer that specifies the status of an HTTP request. All browsers fire the **readystatechange** event when the state has changed to the value 4 and the server's response is complete.

Constant	Value	Meaning
UNSENT	0	open( ) has not been called yet
OPENED	1	open( ) has been called
HEADERS_RECEIVED	2	Headers have been received
LOADING	3	The response body is being received
DONE	4	The response is complete

```
var request = new XMLHttpRequest();
request.open("GET", url);
request.onreadystatechange = function() {
// Request is compete and was successful
    if (request.readyState === 4 &&
        request.status === 200) {
        // Handle the response
    }
};
request.send(null);
```

- To listen for readystatechange events, use **addEventListener**() or set the **onreadystatechange** property of the XMLHttpRequest object to your event handler function.

Request: uploading a file

Back to XMLHttpRequest

- When the user selects a file through an **<input type="file">** element, the form will send the content of that file in the body of the POST request it generates.

```
// Find all <input type="file"> elements with a data-uploadto attribute
// and register an onchange handler so that any selected file is
// automatically POSTED to the specified "uploadto" URL. The server's
// response is ignored.
whenReady(function() {                          // Run when the document is ready
    var elts = document.getElementsByTagName("input"); // All input elements
    for(var i = 0; i < elts.length; i++) {              // Loop through them
        var input = elts[i];
        if (input.type !== "file") continue;  // Skip all but file upload elts
        var url = input.getAttribute("data-uploadto"); // Get upload URL
        if (!url) continue;                   // Skip any without a url

        input.addEventListener("change", function() {  // When user selects file
            var file = this.files[0];         // Assume a single file selection
            if (!file) return;                // If no file, do nothing
            var xhr = new XMLHttpRequest();   // Create a new request
            xhr.open("POST", url);            // POST to the URL
            xhr.send(file);                   // Send the file as body
        }, false);
    }
});
```

Multipart/form-data requests

- A special content-type is known as "**multipart/form-data**".
- The **FormData** API makes it simple to send multipart requests.
  - First, create a FormData object with the FormData() constructor and
  - Then call the append() method as many times as necessary to add the individual "parts"

```
function postFormData(url, data, callback) {
    if (typeof FormData === "undefined")
        throw new Error("FormData is not implemented by browser");

    var request = new XMLHttpRequest();            // New HTTP request
    request.open("POST", url);                     // POST to the specified url
    request.onreadystatechange = function() {      // A simple event handler.
        if (request.readyState === 4 && callback)  // When response is complete
            callback(request);                     // ...call the callback.
    };
    var formdata = new FormData();
    for(var name in data) {
        var value = data[name];
        if (typeof value === "function") continue; // Skip methods
        // Each property becomes one "part" of the request.
        // File objects are allowed here
        formdata.append(name, value);              // Add name/value as one part
    }
    // Note that send automatically sets
    // the Content-Type header when you pass it a FormData object
    request.send(formdata);
}
```

XMLHttpRequest: Decoding response

- Proper **decoding** of a server's response assumes that the server sends a "Content-Type" header with the correct **MIME** type for the response.

- **For a textual response**, with a MIME type like "text/plain", "text/html", or "text/css", it can be retrieved with the **responseText** property of the XMLHttpRequest object.
  - If the server sends **structured data**, such as an object or array, as its response, it might transmit that data as a **JSON-encoded** string. It can be retrieved with the **responseText** property, which is then passed to **JSON.parse**().

- **For an XML or XHTML response**, it can be retrieved through the **responseXML** property. The value of this property is a **Document** object.

XMLHttpRequest: retrieving response

- Decoding Example

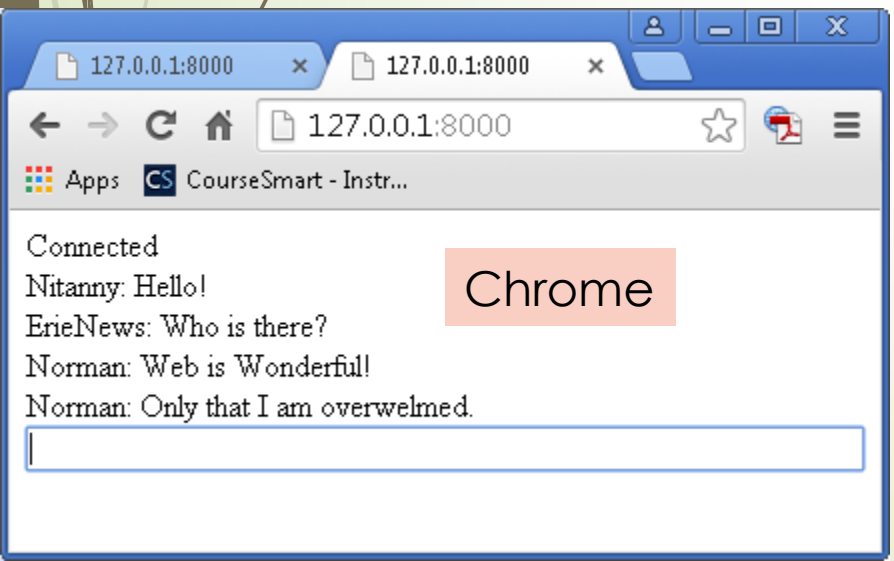
```
1  // Issue an HTTP GET request for the contents of the specified URL.
2  // When the response arrives, pass it to the callback function as a
3  // parsed XML Document object, a JSON-parsed object, or a string.
4  function get(url, callback) {
5      var request = new XMLHttpRequest();          // Create new request
6      request.open("GET", url);                    // Specify URL to fetch
7      request.onreadystatechange = function() {    // Define event listener
8          // If the request is compete and was successful
9          if (request.readyState === 4 && request.status === 200) {
10             // Get the type of the response
11             var type = request.getResponseHeader("Content-Type");
12             // Check type so we don't get HTML documents in the future
13             if (type.indexOf("xml") !== -1 && request.responseXML)
14                 callback(request.responseXML);              // Document response
15             else if (type === "application/json")
16                 callback(JSON.parse(request.responseText)); // JSON response
17             else
18                 callback(request.responseText);             // String response
19         }
20     };
21     request.send(null);                          // Send the request now
22 }
```

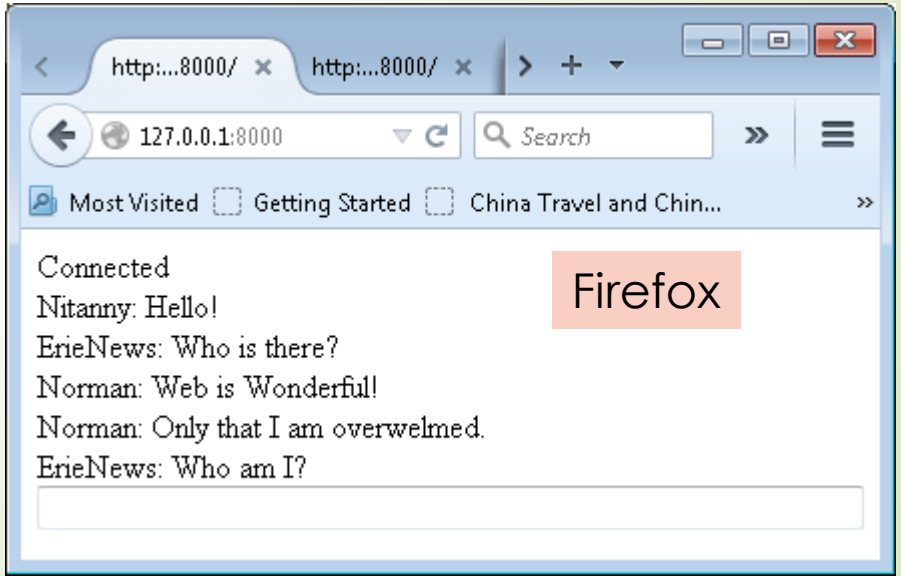
Comet & EventSource API

- **Comet** is the reverse of Ajax
  - In Ajax, the client "pulls" data from the server.
  - With Comet, the server "pushes" data to the client. That is, it is the **web server that initiates** the communication, asynchronously sending messages to the client
- A transport mechanism for Comet requires
  - the client establishes a connection to the server (using an Ajax transport);
  - the server keeps this connection open so that it can send asynchronous messages over it.
- The "Server-Sent Events" in HTML5 defines a simple **Comet API** in the form of an **EventSource** object, which is supported in Chrome, Firefox, and Safari. It is currently **not** supported in **IE**, but can be emulated by XMLHttpRequest.

EventSource: Chat Client

- In a Web browser, when a client requests the root URL "**/**", the server sends the **chatclient.html**
- When creating an Eventsource object, a client makes a **GET** request for the URL "**/chat**". Once the server receives such a request, it saves the response stream in an array and keeps that connection open.
- When a client makes a **POST** request to "**/chat**", it uses the body of the request as a chat message, which is prefixed with the Server-Sent Events "**data:**" prefix, and written to all the open response streams.





EventSource: Chat

Simply pass a URL to the EventSource() constructor and then listen for message events on the returned object

```
<script>
window.onload = function() {
    var nick = prompt("Enter your nickname");      // Get user's nickname
    var input = document.getElementById("input"); // Find the input field
    input.focus();                                 // Set keyboard focus
    // Register for notification of new messages using EventSource
    var chat = new EventSource("/chat");
    chat.onmessage = function(event) {             // When a new message arrives
        var msg = event.data;                      // Get text from event object
        var node = document.createTextNode(msg);   // Make it into a text node
        var div = document.createElement("div");   // Create a <div>
        div.appendChild(node);                     // Add text node to div
        document.body.insertBefore(div, input);    // And add div before input
        input.scrollIntoView();                    // Ensure input elt is visible
    }
    // Post the user's messages to the server using XMLHttpRequest
    input.onchange = function() {                  // When user strikes return
        var msg = nick + ": " + input.value;       // Username plus user's input
        var xhr = new XMLHttpRequest();            // Create a new XHR
        xhr.open("POST", "/chat");                 // to POST to /chat.
        xhr.setRequestHeader("Content-Type",       // Specify plain UTF-8 text
                             "text/plain;charset=UTF-8");
        xhr.send(msg);                             // Send the message
        input.value = "";                          // Get ready for more input
    }
};
</script>
<!-- The chat UI is just a single text input field -->
<!-- New chat messages will be inserted before this input field -->
<input id="input" style="width:100%"/>
```

Chatserver.js

```
7  var server = new http.Server();
8  server.on("request", function (request, response) {
9      var url = require('url').parse(request.url);
10     if (url.pathname === "/") {  // A request for the chat UI
11         response.writeHead(200, {"Content-Type": "text/html"});
12         response.write(clientui);
13         response.end();
14         return;
15     }
16     else if (url.pathname !== "/chat") {
17         response.writeHead(404);
18         response.end();
19         return;
20     }
21     if (request.method === "POST") {
22         request.setEncoding("utf8");
23         var body = "";
24         request.on("data", function(chunk) { body += chunk; });
25         request.on("end", function() {
26             response.writeHead(200);   // Respond to the request
27             response.end();
28             message = 'data: ' + body.replace('\n', '\ndata: ') + "\r\n\r\n";
29             clients.forEach(function(client) { client.write(message); });
30         });
31     }
32     else { //handling a client's new EventSource("/chat");
33         response.writeHead(200, {'Content-Type': "text/event-stream" });
34         response.write("data: Connected\n\n");
35         request.connection.on("end", function() {
36             clients.splice(clients.indexOf(response), 1);
37             response.end();
38         });
39         clients.push(response);
40     }
41 });
42 server.listen(8000);
```

The client initiates a connection to the server (when it creates the EventSource object)
The server keeps this connection open.
When an event occurs, the server writes lines of text to the connection.