AJAX

AJAX (originally an acronym for Asynchronous JavaScript and XML) allows a client-side JavaScript program to exchange data with a webserver. Typically the data sent will be in the form of an XML-file, although AJAX can be used to send text files as well.

Using AJAX, our client-side program will send a standard HTTP request to our webserver. Our program will setup a callback function which will be called when the webserver has responded with the data requested. Once received our callback function will process the data, typically modifying the webpage using the standard dynamic content techniques described in the previous handout.

XMLHttpRequest

The XMLHttpRequest object is the key to using AJAX. This object was first defined by Microsoft for use in Internet Explorer. It has since been adapted by other web browsers, and a proposed standard version is being drafted by the W3C.

Creating an XMLHttpRequest Object

In order to use AJAX, we'll need to create an XMLHttpRequest object. Here's the code for this:

```
requestObj = new XMLHttpRequest();
```

Setting Up a Request using XMLHttpRequest

Once we've created our XMLHttpRequest object, we're ready to put together a request for our webserver. We can use an HTTP GET, POST, or HEAD request (additional HTTP requests may be supported depending on the browser).

We'll need to take several steps in order to get our request ready. First, we tell the XMLHttpRequest object, which of our JavaScript functions to use as a callback when we receive a response back from the webserver. Next we put together our actual request to the webserver. Finally we actually send the request to the webserver.

Suppose, for example, we have a function named handleResponse that we want to execute when the webserver sends our data. We would assign the handler function to our XMLHttpRequest object as follows:

```
requestObj.onload = handleResponse;
```

or using the more formal:

```
requestObj.addEventListener("load", handleResponse,null);
```

We'll take a look in a moment what our callback function actually needs to do when it executes.

Next we setup our request to the webserver using the XMLHttpRequest object's open and send methods. The open method takes three parameters—first, the type of request (e.g., GET, POST, HEAD),[1] second the actual URL requested, and third a boolean telling the system whether or not the call is asynchronous (that is whether the system should wait until the data is received or should continue regular operation until the data is received). The send method is used to send information in the request body as associated with a POST request. If you aren't using a POST call send with a null parameter.

Here is an example requesting the file info.xml, were we assume info.xml is located in the same directory as the current HTML file (the one whose JavaScript we are executing):

```
requestObj.open('GET', 'info.xml', true);
requestObj.send(null);
```

The URL requested can be a relative or absolute HTTP reference. Here we are requesting a specific file on the Stanford webserver:

```
requestObj.open('GET', 'http://www.stanford.edu/dept/cs/info.xml', true);
requestObj.send(null);
```

Warning: If the URL requested isn't on the same webserver as the webpage making the request, you'll have to do some additional work. For our class we'll be using the same webserver for the requesting page and the requested URL. See the W3C's "Cross-Origin Resource Sharing" document for details on how to handle different servers.

http://www.w3.org/TR/cors/

Note: While we can create a synchronous HTTP request using the third parameter of the open call, this is not recommended. The user will not be able to interact with your web page while waiting for the web server to return its data.

Responding to Server Data

Our callback function will get called once the data has downloaded. So, once the transfer has been completed, how do we get a hold of the information? Our actual data is in one of two places. If we've requested an XML document, we can access its document object model in the XMLHttpRequest object's responseXML property. If the file sent is a text file, it can be found in the object's responseText property.

If the object returns a responseXML, the object will be in the form of a DOM tree. You can use the functions we discussed in the Dynamic Content lecture in order to access the nodes in the tree. For example, the following handler code retrieves an array of all the elements in the XML file with tag name "title":

```
function handleResponse() {
    requestObj.responseXML.getElementsByTagName("title");
    …
}
```

[1] For those of you unfamiliar with HTTP, we can send several different types of requests to a webserver. A GET request as the name implies requests a file from the webserver. A POST sends information to the webserver and receives a file in response. A HEAD is the same as a GET, except the actual file isn't sent, instead only the header accompanying a regular GET response is sent. This header contains information about the file (such as modification date and size).

Alternative Actions

While we certainly hope our request is properly carried out, there are some additional callbacks functions we can set to monitor progress or to handle errors. To do this we can call addEventListener and use the events: loadstart, progress, load, loadend, error, or abort. Note these may not be supported on all web browsers. The meanings are as follows:

- loadstart – as its name implies this will get called as soon as the load begins.
- progress – this is potentially called repeatedly and if supported can be used to determine progress rate.
- error – this is called if the load fails.
- abort – this is called if the request is aborted.
- load – setting this is the same as setting the onload property.
- loadend – called on successful completion of load, on error, or on abort.

You may assign these like this:

```
requestObj = new XMLHttpRequest();

requestObj.addEventListener("error",errorHandler,false);
```

On older web browsers, none of these events (including load) is supported and you'll need to revert to an older property called onreadystatechange, which with a bit more work, will fulfil the same purpose as all the above properties.

Additional Information

The current W3C draft specifies a timeout property on the XMLHttpRequest object. Setting this property will tell the browser that we are only willing to wait for the given number of milliseconds before terminating the request.

A second property ontimeout specifies a function to call if the timeout occurs.

These properties are relatively late additions to the W3C draft, so if you plan to use them, test thoroughly on whatever web browsers your website is planning to support.