Overview

- Asynchronous JavaScript and XML (AJAX)
  - The XMLHttpRequest Object
- Browsers, JavaScript, & the DOM
  - DOM vs InnerHtml manipulations
  - Node and Node types
  - JavaScript methods and iteration
- JSON as a payload handler
  - Security issues with JSON in AJAX
- JSON in Javascript
- JSON in Java
Asynchronous Javascript and XML (AJAX)

- The key technology for AJAX was created by Microsoft in 1999 – a JavaScript object that could send a request to a server and handle a response.
- The name results from two things
  - Asynchronous JavaScript refers to the use of JavaScript to send a message and handle the response asynchronously – whenever it comes back
  - XML was initially proposed for use in handling the data transferred and XML also provides the Document Object Model (DOM) which is manipulated in the browser as a directed acyclic graph – a tree structure

The AJAX Object

- For non IE browsers create an XMLHttpRequest object:
  
  ```javascript
  var AJAXobj = new XMLHttpRequest();
  ```

- For IE, use ActiveXObject:
  
  ```javascript
  var AJAXobj  = new ActiveXObject("Microsoft.XMLHTTP");
  ```

- Note: this also means that if users have ActiveX objects disabled in Internet Explorer, they will be unable to use XMLHttpRequest even if JavaScript is enabled.
- The code fragment to create the AJAXObj normally looks something like this:
  
  ```javascript
  if (window.XMLHttpRequest)
  {
    // code for IE7+, Firefox, Chrome, Opera, Safari
    AJAXobj=new XMLHttpRequest();
  }
  else
  {
    // code for IE6&E5
    AJAXobj=new ActiveXObject("Microsoft.XMLHTTP");
  }
  ```

XMLHttpRequest Methods

- `open("HTTP method", "URL", [syn/asyn, [uname, pass]]):`
  - Assigns HTTP method, destination URL
  - Syn/asyn is an optional boolean – default true = asynchronous
  - Uname and pass are optional and used for authentication
- `setRequestHeader("label", "value")`: Sets Request Headers before sending
- `send(content)`: Sends request including string or DOM object data
- `abort()`: Terminates request
- `getAllResponseHeaders()`: Returns headers (labels + values) as a string
- `getResponseHeader("header")`: Returns value of a given header
XMLHttpRequest Properties

• onreadystatechange
   • The JavaScript event handler that fires at each state change
• readyState – current status of request
   • 0 = uninitialized (The request is not initialized)
   • 1 = loading (The request has been set up)
   • 2 = loaded (The request has been sent)
   • 3 = interactive (The request is in process)
   • 4 = complete (The request is complete)

XMLHttpRequest Properties (cont)

• status
   • HTTP Status returned from server: 200 = OK
• statusText
   • Status text returned from server
• responseText
   • String version of data returned from the server
• responseXML
   • XML element node returned from the server

Processing Responses

• Given an AJAXObj, the most direct way to handle a response is to define an anonymous function that is activated every time the “ready state” of the request changes:
  ```javascript
  AJAXObj.onreadystatechange = function() {
    if (self.xmlHttpReq.readyState == 4) {
      do something with AJAXObj.responseText;
    }
  }
  ```
• There are four ready states that relate to the initiation, processing, and completion of the request. "4" indicates that the response is completed. This code does not address checking the status of the server result – e.g. 200, 403, etc.
• Also, keep in mind that there are a variety of response objects besides responseText
Browsers, JavaScript & the DOM

Browser and DOM

- Browsers maintain an object representation of the documents being displayed
  - In the form of Document Object Model (DOM)
  - It is readily available as document JavaScript object
- APIs are available that allow JavaScript code to modify the DOM programmatically

DOM APIs vs. innerHTML

- Assume message is something like:
  - `<p>some message to be posted</p>`
- And that the user message element is a div
- Using innerHTML is easy: Sets or retrieves the HTML between the start and end tags of the object

```javascript
function setMessageUsingInnerHTML(id, message) {
    var Elem = document.getElementById(id);
    Elem.innerHTML = message;
}
```
DOM APIs vs. innerHTML

- DOM APIs provide a means for JavaScript code to navigate/modify the content in a page more intimately
- In this case assume message is a plain text string.

```javascript
function setMessageUsingDOM(id, message) {
  var Elem = document.getElementById(id);
  var para = document.createElement("p");
  var messageBody = document.createTextNode(message);
  para.appendChild(messageBody);
  if (Elem.childNodes[0]) {
    Elem.replaceChild(para, Elem.childNodes[0]);
  } else {
    Elem.appendChild(para);
  }
}
```

JavaScript and the DOM

- JavaScript includes extensive support for manipulating DOM objects
- It is possible to access nodes in the tree a variety of ways, and then once accessed grab hold of other nodes
- Common way to access a node are:
  - `document.getElementsByTagName("p")`
  - `document.getElementById("id3")`
- Once a node or set of nodes is obtained, you can check its type or value – see next slide
  - There may be multiple text nodes around an element – extra white space
- Keep in mind that there is a good online reference is:
  - `http://www.w3schools.com/dom/

Node Type Return Values

<table>
<thead>
<tr>
<th>Node type</th>
<th>nodeName</th>
<th>nodeValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document(9)</td>
<td>#document</td>
<td>null</td>
</tr>
<tr>
<td>DocumentFragment</td>
<td>#document fragment</td>
<td>null</td>
</tr>
<tr>
<td>DocumentType</td>
<td>doctype name</td>
<td>null</td>
</tr>
<tr>
<td>EntityReference</td>
<td>entity reference name</td>
<td>null</td>
</tr>
<tr>
<td>Element(1)</td>
<td>element name</td>
<td>null</td>
</tr>
<tr>
<td>Attr(2)</td>
<td>attribute name</td>
<td>attribute value</td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td>target</td>
<td>content of node</td>
</tr>
<tr>
<td>Comment(8)</td>
<td>#comment</td>
<td>comment text</td>
</tr>
<tr>
<td>Text(3)</td>
<td>#text</td>
<td>content of node</td>
</tr>
<tr>
<td>CDATASection(4)</td>
<td>#CDATA-section</td>
<td>content of node</td>
</tr>
<tr>
<td>Entity</td>
<td>entity name</td>
<td>null</td>
</tr>
<tr>
<td>Notation</td>
<td>notation name</td>
<td>null</td>
</tr>
</tbody>
</table>
Node Properties

- Examples of DOM node properties include:
  - x.nodeName - the name of x
  - x.nodeValue - the value of x
  - x.parentNode - the parent node of x
  - x.childNodes - the child nodes of x
  - x.attributes - the attributes nodes of x
- Keep in mind that some properties are null for some nodes and that the return value may be a node or a node set.

Iterating over a Nodelist

- Using getElementsByTagName
  - x=document.getElementsByTagName("title");
  - for (i=0;i<x.length;i++)
    - { document.write(x[i].innerHTML); document.write("<br />"); }

- When using childNodes
  - x=document.getElementById("XXX").childNodes;
  - for (i=0;i<x.length;i++)
    - { if (x[i].nodeType==1) //Process only element nodes (type 1)
      - document.write(x[i].nodeName); document.write("<br />"); }

- Note that node types equate to numbers:
  1=ELEMENT_NODE,
  2=ATTRIBUTE_NODE,
  3=TEXT_NODE,
  4=CDATA_SECTION_NODE,
  5=ENTITY_REFERENCE_NODE,
  6=ENTITY_NODE,
  7=PROCESSING_INSTRUCTION_NODE,
  8=COMMENT_NODE,
  9=DOCUMENT_NODE
Document Node Methods

- We have already seen some document node methods:
  - `getElementById(id)`
  - `getElementsByTagName()`
- There are several other methods including:
  - `createComment()`
  - `createElement()`
  - `createTextNode()`
  - `createAttribute(name)`

Element Node Methods

- The element node methods – which are extensive – include a variety of DOM construction functions:
  - `appendChild()`
  - `cloneNode()`
  - `hasChildNodes()`
  - `insertBefore()`
  - `removeChild()`
  - `replaceChild()`
  - `getAttribute()`
  - `removeAttribute()`
  - `setAttribute()`
Basics of JavaScript Objects

- A JavaScript object is very simple to create.
- It is simply an associative array.
- To create one manually, one would enter something like the following:
  ```javascript
  var myobj = { "a":10, "b":"Hello" };  
  ```
- The elements of the array consist of a set of name value pairs between curly braces {}.
- This results in myobj.a having the value 10.

More Complex Javascript Objects

- More complex JavaScript objects can also be created. For example:
  ```javascript
  var x =  
  { "firstName": "John",  "lastName": "Smith",  
    "address": {  
      "streetAddress": "21 2nd Street",  
      "city": "New York",  "state": "NY", "postalCode": 10021 }, 
    "phoneNumbers": [  
      "212 555-1234",  "646 555-4567" ]  
  }  
  ```
- Now we have things like:
  ```javascript
  x.firstname = "John"  
  x.address.state="NY"  
  x.phoneNumbers[0]="212 555-1234"  
  ```

JSON Libraries

- JSON stands for JavaScript Object Notation.
- It is a simple hash that can be used to store and represent objects of arbitrary complexity.
- It has been suggested by a number of people as an alternative to an XML payload in AJAX.
- JSON "adapters" are available for most languages.
- See www.json.org
- Org.json is the Java package
- Json2.js is the JavaScript package
Creating a JSON Object and "Stringifying" it

var request;
if ((myform=document.getElementById(value))!= null){
  var num = myform.elements.length;
  for(i=0; i < num; i++) {
    // for this simple case, we only process text inputs
    if (myform.elements[i].type != "text") continue;
    var name = myform.elements[i].name;
    var value = myform.elements[i].value;
    request[name] = value;
  }
} Else {/*problem*/
  // now we use Stringify to turn the object into a string
  Message = JSON.stringify(request);//this is from JSON2.js
}

Converting a Stringified JSON Object back to an Object

• The code below assumes we have returned a Stringified JSON object:
• eval converts the string to an object
• We assume the names of the pieces are the ids of the elements where values are to be inserted
var JSONresp = eval(JSONObj.response)
for(i=0;i < JSONresp.length; i++) {
  var el = document.getElementById(JSONresp[i].name)
  el.innerHTML = JSONresp[i].value;
}

Java: Reading Stringified JSON

• Assumes we are using json.org
BufferedReader in = request.getReader();
amtread = in.read(cbuf);// this is too simplistic
if(cbuf[amtread]!=0);
String invaluce = new String(cbuf,0,amtread);
try{
  jo=new JSONObject(invalue)
  names = JSONObject.getNames(jo);
  for (int j =0; j<names.length;++j)
    {values[j]=jo.getString(names[j]);}
Java: Writing an Object to String

```java
PrintWriter out = response.getWriter();
jo2=new JSONObject();
for (int j =0; j<names.length;j++)
    {jo2.append(names[j], values[j]);}
out.println(jo2.toString());
```

AJAX Security: Server Side

- AJAX-based Web applications use the same server-side security schemes of regular Web applications
  - You specify authentication, authorization, and data protection requirements in your web.xml file (declarative) or in your program (programmatic)
- AJAX-based Web applications are subject to the same security threats as regular Web applications
  - Cross-site scripting
  - Injection flaw
- JSON represents a potential security risk which is mitigated by using the JSON scripts

AJAX Security: Client Side

- JavaScript code is visible to a user/hacker
  - Hacker can use the JavaScript code for inferring server side weaknesses
- JavaScript code is downloaded from the server and executed ("eval") at the client
  - Can compromise the client by mal-intended code
- Downloaded JavaScript code is constrained by sand-box security model
  - Can be relaxed for signed JavaScript