Overview

- What AJAX is all about
- XMLHttpRequest Object
- Callback function
- Dom manipulation
- Firefox debugging Tools
  - Firebug
  - Dom inspector
- Some Complications and Solutions
  - Browser differences
  - Toolkits

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
  - Javascript is an object based scripting language that uses the XML DOM as the basis for operation
  - Asynchronous JavaScript refers to the use of JavaScript to send a message and handle the response asynchronously whenever it comes back
  - XML was initially proposes 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&ES
    AJAXobj = new ActiveXObject("Microsoft.XMLHTTP");
  }
  ```

Request Methods

- Simple requests
  - A simple GET request:
    ```javascript
    requester.open("GET", "/query.cgi?name=Bob&email=bob@example.com", true);
    requester.send(null);
    ```
  - A simple POST request:
    ```javascript
    requester.open("POST", "/query.cgi", true);
    requester.send("name=Bob&email=bob@example.com");
    ```
- In reality, requests can be much more complex and may involve the construction of XML structures or JSON objects

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. At this point, we have not addressed the status of the server result – e.g. 200, 403, etc.
- There are also a variety of response objects besides text
Simple Query & Update

function getquerystring() {
}

function updatepage(str){
    var el = document.getElementById("result");
    el.innerHTML = str;
}

Putting It All Together
Main Function

• We define four separate functions. The main function controls functions to construct an AJAXObj, construct the message, and indicate how the response should be handled:

    var AJAXObj

    function makeAJAXRequest() {
        AJAXObj = getAJAXObject();
        AJAXObj.open('POST', 'http://www.server/ServerProgram', true);
        Req.setRequestHeader('Content-Type', 'text/xml');
        Req.onreadystatechange = processResponse();
        Req.send(getXMLString());
    }

Putting It All Together
Construction Function

• The construction function takes into account cross browser issues and may also note the browser type for other uses.

    var browserClass;

    function getAJAXObject(){
        browserClass = navigator.userAgent; // this is normally simplified
        if (window.XMLHttpRequest)
            AJAXObj = new XMLHttpRequest();
        else
            AJAXObj = new ActiveXObject("Microsoft.XMLHTTP");
        return AJAXObj;
    }
Putting It All Together

Request Construction

- Any number of different strategies might be used to construct messages of many different forms. This example constructs an XML fragment from a form — and assumes all form elements are simple text input elements. (The JavaScript escape function takes care of the URL encoding.)

```javascript
function getXMLString(fname) {
    var myform = document.forms[fname];
    qstr = "<form>";
    for (var i=0;i<myform.elements.length;i++) {
        qstr += "<" + myform.elements[i].name + "">" + escape(myform.elements[i].value) + "</" + myform.elements[i].name + ">";
    }
    qstr += "</form>";
    return qstr;
}
```

Response Processor

- There are any number of forms responses can take and any number of ways that they can be handled. Here we assume it is an html fragment that can simply be appended to an object in the document prepared for it.

```javascript
var AJAXObj;

function processResponse() {
    var message = AJAXObj.responseText;
    var targetel = document.getElementById("responsediv");
    targetel.innerHTML = message;
}
```

The Server Side

- AJAX is said to be server agnostic — it really doesn’t matter what kind of code sits on the other side.
- For a GET, the parameters will be URL encoded and attached to the request:
  - `http://some.server/someprogram?name=value`
- For a post, the same query string, again URL encoded will be put in the message
- Any program acting on the server side will simply use its standard mechanism for accessing the parameters of a request as shown on the next slide.
Access to AJAX Request in Java Servlets

protected void processRequest(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    resp.setContentType("text/plain;charset=UTF-8");
    PrintWriter out = resp.getWriter();
    try {
        String id = req.getParameter("id");
        if (dbmscheck(id)) {
            out.println("This user has been validated");
        } else {
            out.println("This user is not validated");
        }
    } finally {
        out.close();
    }
}

Debugging Tools for Firefox

- There are numerous tools for debugging JavaScript — JavaScript Console, Venkman JavaScript Debugger, FireBug, etc.
- Firebug provides the most comprehensive set of features which include:
  - HTML explorer
  - CSS review
  - JavaScript Debugger
  - DOM Modeling
  - Network Communications
  - Cookies
- A couple of these features are shown on the next slide

FireBug Components

- JavaScript Debugger
- HTML Inspector
- Network Traffic
Complications and Solutions

• One of the biggest issues in the use of AJAX is how data is transmitted.
  • When the data is simple, simple strings are more than satisfactory in requests
  • When both the client – the HTML page – and the server – e.g. a servlet – are prepared to do DOM processing, XML is a good choice.
  • If there is a lot of numeric data, and DOM processing isn’t useful or needed, the JavaScript Object Notation or JSON becomes an interesting choice.
  • These last two solutions are briefly described on the next few slides

Using XML for Data Transfer

• There are a number of options for composing and interpreting XML.
  • It can be done in brute force fashion – by simply building a valid string
  • It can be done using DBMS XML construction code
  • It can be done by constructing a serializing a DOM object
  • There are API’s in most languages and the JavaScript and Java API’s are very similar. The next page shows a sample in Java.

```
processRequest(HttpServletRequest req, HttpServletResponse resp)
{
  try {
    BufferedReader br = req.getReader();
    StringBuffer sb = new StringBuffer();
    String line = null;
    while((line=br.readLine())!=null) {sb.append(line);}
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    InputSource is = new InputSource( new StringReader( sb.toString() ) );
    Document d = builder.parse( is );
    NodeList list = d.getElementsByTagName("sometag"); // this is much more complex
    Element e = list[0];
    String id = e.getChild[0].getValue();
    if (dbmscheck(id))
      {out.println("This user has been validated");}
    else
      {out.println("This user is not validated");}
  } catch (Exception E){// really bad, but brief}
}
```
Building and Writing an XML Document

```java
processRequest(HttpServletRequest req, HttpServletResponse resp) {
    try {
        resp.setContentType("text/xml;charset=UTF-8");
        PrintWriter out = response.getWriter();
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document d = builder.newDocument();
        Element x = d.createElement("sometag");
        Text y = d.createTextNode("some text");
        x.appendChild(y);
        d.appendChild(x);
        TransformerFactory tfac = TransformerFactory.newInstance();
        Transformer tf = tfac.newTransformer();
        StringWriter sw = new StringWriter();
        StreamResult result = new StreamResult(sw);
        DOMSource source = new DOMSource(d);
        tf.transform(source, result);
        out.println(sw.toString());
    } catch (Exception E) { // really bad, but brief }
}
```

2013/11/5 IS 2870: Web Technology and Standards

Building and Sending a JSON Object

- A JSON Object can be constructed manually:
  ```javascript
  var jobj = {"name1": "value1 (a string)", "name2": 13, "name3": 65.3};
  ```
- They can be nested and arbitrarily complex.
- They can also be constructed through code as shown here:
  ```javascript
  var myform=document.getElementById(formid);
  for (i=0;i<myform.length;i++){
    if (myform[i].type == "text"){
      jobj[myform[i].name]=myform[i].value;
    }
  }
  ```
- Once Constructed, they can be send using AJAX as follows:
  ```javascript
  xhr.open("POST", target, true);
  xhr.send(JSON.stringify(jobj));
  ```
- The JSON.stringify function comes from JSON.org and is included by including the file json2.js in your html/jsp page.

Reading JSON in Java

- To process json in Java you need to include a JSON library/jar file or the json.org package from json.org or sourceforge.net
- Simply read the request using the reader and convert it to the Java equivalent of the original object:
  ```java
  protected void processRequest(HttpServletRequest req, HttpServletResponse resp)
  {
    char[] cbuf = new char[2000];
    int amtread;
    JSONObject jo;
    BufferedReader in = request.getReader();
    amtread = in.read(cbuf);
    cbuf[amtread] = '0';
    String invalue = new String(cbuf,0,amtread);
    try{
      jo = new JSONObject(invalue);
      names = jo.names();
      for (int j =0; j<names.length;j++)
      {values[j]=jo.getString(names[j]);}
    }
    catch(Exception e) { // TOSAVE SPACE }
  }
  ```
Toolkits

• A variety of toolkits have been developed and wrapped around AJAX to make it easier to use.
  • They attempt to overcome issues related to browser incompatibility
  • They attempt to simplify building dynamic capabilities into web pages
• The common toolkits include:
  • Prototype with Script.aculo.us
  • Dojo
  • Direct Web Remoting
  • Yahoo UI Library
  • Google Web Toolkit

DOJO

The Dojo Toolkit

• AN Open Source toolkit written in JavaScript
  • It is a set of JavaScript libraries
  • Available form http://dojotoolkit.org/
• It provides:
  • AJAX-based I/O abstraction (remoting)
  • Backward, forward, bookmarking support
  • Aspect-oriented event system
  • Markup-based UI construction through widgets
  • Widget prototyping
  • Animation
**Dojo Toolkit Libraries**

- dojo.io: AJAX-based communication with the server
- dojo.event: unified event system for DOM and program events
- dojo.lang: utility routines to make JavaScript easier to use
- dojo.string: String manipulation routines
- dojo.dom: DOM manipulation routines
- dojo.style: CSS Style manipulation routines
- dojo.html: HTML specific operations
- dojo.date: Date manipulation
- dojo.logging.Logger: Logging library
- dojo.profile: JS Profiler
- dojo.regexp: Regular Expression generators
- dojo.dad: Drag and Drop

---

**dojo.io.bind()**

- The dojo.io.* namespace contains generic APIs for
  - doing network I/O
  - dojo.io.bind() hides low-level XMLHttpRequest operations
- A generic asynchronous request API that wraps multiple transport layers
  - queues of iframes, XML HTTP, mod_pubsub, LivePage, etc. (For now, only XMLHTTP will ever be chosen since no other transports are rolled in)
  - Dojo attempts to pick the best available transport for the request at hand

---

**dojo.io.bind() Syntax**

```javascript
// Make a request that returns raw text from a URL
dojo.io.bind(
    // URL to make a request to
    url: "http://foo.bar.com/something",
    // Callback function to execute upon successful response load:
    function(type, data, evt) {
        /*do something w/ the data*/
    },
    // Content or other members to load data
    Content: {name: value, name2, value2},
    // Type of data that is returned
    mimetype: "text/plain"
    // More options
);
```
Use dojo.io.bind()

- Modify index.jsp file in "Data Validation with AJAX" sample application to use dojo.io.bind()
- Don't need to do the low-level XMLHttpRequest handling anymore
- Don't need to check the status and readyState anymore
- Several convenience forms for loading data

Use dojo.io.bind() Cont'

```javascript
function validateUserId() {
  if (!target) target = document.getElementById("userid");
  var sampleFormNode = document.getElementById("submit_btn");
  // Use dojo.io.bind() for remoting
  dojo.io.bind(
    // URL - destination you want to send your request
    url: "validate?id=" + escape(target.value),
    // Callback function that will be invoked asynchronously
    load: function(type, data, evt){ processRequest(data);},
    // Content contains the data in several forms - simplest is
    content: {name1: value, name2: value},
    // Error handler function that will be invoked in case of an error
    error: function(type, error){ alert("error"); },
    // several other possible members exist
  );
}
```

Use DOM manipulation methods of Dojo toolkit

```javascript
// Modify function validateUserID
function validateUserID() {
  // if (!target) target = document.getElementById("userid");
  if (!target) target = dojo.byId("userid");
  // if (target) target = dojo.byId("userid");
  if (target) target = dojo.byId("userid");
  // var sampleFormNode = document.getElementById("submit_btn");
  var sampleFormNode = dojo.byId("submit_btn");
  // var sampleFormNode = dojo.byId("submit_btn");

  // Use dojo.io.bind() for remoting
  dojo.io.bind(
    ...
  );
}
```