AJAX & XML

Chapter 12

- Many interesting websites revolve around a ton of data.
- The data has many formats: images, videos, text, XML, HTML...
- **Ajax** - a way to dynamically update a page using data fetched from a server. Short for **A**synchronous **J**avaScript **a**nd **X**ML.
- Ajax can only fetch files from it's own server.
- Debugging HTTP requests can be difficult. Bring up the JavaScript debugger in your browser and click on the Network tab.

- The general idea of Ajax is:
  - User's browser creates a JavaScript object called an **XMLHttpRequest**.
  - The XMLHttpRequest requests information from the web server.
  - Data is sent back to the browser in XML format.
  - JavaScript injects the XML data into the web page and displays it to the user.
- Ajax is a different way of the user communicating with the web server.

Traditional

- When you click on a link, your browser fetches a new web page and displays it.
- When you click on another link, your browser fetches a new web page.
- This process is repeated. . . This is called **synchronous communication**. In other words, you must wait for the page to load before you can do anything.

Ajax

- With Ajax, the user's interaction with a web page triggers a request to receive a small amount of data.
- When the data arrives back from the web server, the user's browser remains on the same page and JavaScript updates the page.
- This is called **asynchronous communication** because multiple things can happen at the same time without needing to wait.
- There are many applications for Ajax: real-time form validation, auto-completion, load on demand, better user interface (trees, menus, progress bars), refreshing data (e.g. sports scores, stock quotes, weather), partial submission of forms and making web applications feel like desktop applications (even "complete" operating systems).

Ajax is not an trivial concept. We will go through the process in small steps.

1. Synchronous JavaScript with text-only data.
2. Asynchronous JavaScript with text-only data.
3. Asynchronous JavaScript with text-only data, using Prototype.
4. Asynchronous JavaScript with XML data.

- Let's assume we have a web page with button and a text area.
- When the button is pressed, the contents of some text file is shown in the text area.

```
//generic synchronous ajax code
var example = new XMLHttpRequest();
example.open("GET", "someURL", false);
example.send(null);

//the web request is completed, text is stored in
//example.responseText
```

- In the HTML file:

<textarea id="output" rows="4" cols="20">Some text</textarea>
<button id="load">Load it!</button>

- In the JavaScript file:

```
window.onload = function() {
    document.getElementById("load").onclick = loadClick;
};

function loadClick(){
    var ajax = new XMLHttpRequest();
    ajax.open("GET", "numbers.txt", false);
    ajax.send(null);
    document.getElementById("output").value = ajax.
        responseText;
}
```

- The first 3 lines of loadClick are so common that you'll likely want to make a helper function that returns the responseText.

- The example on the previous page was synchronous because *ajax.send(null)* waits for the request to finish before returning. It waits because of the false here: ajax.open("GET", "numbers.txt", **false**);
- The next line of code will not execute until the request is finished.
- Programming is easier but the user experience suffers.

- As with the majority of web programming, debugging is painful.
- The XMLHttpRequest may fail.
- An HTTP code is placed in the **status** property. **statusText** is a property that explains the status property.

```
//HTTP error code 200 represents a success
function downloadURL(url){
    var ajax = new XMLHttpRequest();
    ajax.open("GET", url, false); //synchronous
    ajax.send(null);

    if(ajax.status != 200){
        alert("Error with " + url + ": " + ajax.status + " "
            + ajax.statusText);
    }

    return ajax.responseText;
}
```

- The idea behind asynchronous requests is that the code makes the request and doesn't wait for a response.
- Rather, the request is made to the server along with a function that will be called when the request is completed.

```
//generic asynchronous JavaScript request
var example = new XMLHttpRequest();
example.onreadystatechange = function(){
    if(example.readyState == 4){
        //do something with request here
    }
};
ajax.open("GET", "someURL", true); //note true here
ajax.send(null);
```

- The function being called later in the code below is referred to as a **callback**.
- A function declared inside of another funciton is called a **nested function**.
- Nested functions can access all variables from the outer function in which they are declared. This idea is called a **closure**.
- **readyState** goes from 0 (uninitialized) to 1 (setup, not sent) to 2 (sent) to 3 (in progress) to 4 (completed).

```
//generic asynchronous JavaScript request
var example = new XMLHttpRequest();
example.onreadystatechange = function(){
    if(example.readyState == 4){
        //do something with request here
    }
};
ajax.open("GET", "someURL", true); //note true here
ajax.send(null);
```

- downloadURL is modified to handle asynchronous calls.

```
function downloadTextAsync(url, func){
    var ajax = new XMLHttpRequest();
    ajax.onreadystatechange = function(){
        if(ajax.readyState == 4){
            if(ajax.status == 200){
                //call function passed in
                func(ajax);
            } else{
                //something went wrong
                alert(ajax.status + " " +ajax.statusText);
            }
        }
    };
    ajax.open("GET", url, true); //asynchronous
    ajax.send(null);
}
```

- Prototype provides some wrapper objects to make ajax programming a bit simpler. If using this, don't forget to include the prototype.js script.
- A new *Ajax.Request* object is created and the constructor does all of the work.
- Much easier to deal with errors.

```
new Ajax.Request(
    "someURLString",
    {
        option : value,
        option : value,
        ...
        option : value
    }

);
```

- The option : value pairings below are called **anonymous objects**.
- When Prototype calls one of the functions below, the XMLHTTPRequest is passed in as an argument.

```
new Ajax.Request(
    "numbers.txt",
    {
        method : "get",
        asynchronous : true, //not necessary
        onSuccess : someFunction,
        onFailure : someOtherFunction,
        onComplete : thirdFunction,
        onException : someOtherFunction,
        on404 : someOtherFunction
    }

);
```

- A nice feature of Prototype is using the **Ajax.Updater** object.
- It is common to read new text from a server and place it somewhere on the page.
- More wrappers can be found at http://api.prototypejs.org/ajax/Ajax/

```
new Ajax.Updater(
    "idOfElement",
    "someURL",
    {
        option : value,
        option : value,
        ...
        option : value
    }

);
```

- Text data used to be split using whitespace (tabs, line breaks, etc) or some other delimiter.
- Since our splitting was arbitrary, no other software would likely be able to help us break apart the data.
- **XML** - specification for creating markup to store hierarchical data.
- XML describes a tag-based syntax for marking up text but does not specify what tags should and shouldn't be used.
- You've already seen it in this class. Where?

- Think about the following text that is split by a | symbol.
- We would first have to split it up by lines, then split it up by | symbols. If some data is omitted, the split could be more difficult.

Example

David | Furcy | Georgia Tech | 1182
Erik | Krohn | Iowa | 7080
Tom | Naps | Notre Dame | 1388

- XML is like HTML with no pre-defined tags or attributes. You can use whatever tags and attributes you want.
- The previous page's example is shown in some XML form:

Example

```
<?xml version = "1.0" encoding="UTF-8"?>
<people>
    <person school="Georgia Tech" phone="1182">
        David Furcy
    </person>
    <person school="Iowa" phone="7080">
        Erik Krohn
    </person>
    <person school="Notre Dame" phone="1388">
        Tom Naps
    </person>
</people>
```

- A legal XML document must start with an **XML prologue**: <?xml version = "1.0" encoding="UTF-8"?>
- Following the prologue must be the **document** tag. This is the outermost tag. On the previous slide, <people> was the outermost tag.
- **Schema** - an optional document that describes which tags and data are legal in some XML file.
- W3C uses an HTML5 schema to ensure a web page is HTML5 compliant.
- Schemas are specified in many different languages including **Document Type Definition (DTD)** and **W3C XML Schema**.

- Let's say we have a web page setup to pick a school. We would then use ajax to list all professors who went to that school.
- The partial web page would look something like:

```
<div>
    <p>Universities:</p>
    <ul id="university">
        <li>Retrieving</li>
    </ul>
</div>
<div>
    <p>Professors:</p>
    <ul id="professors">
        <li>Select a university</li>
    </ul>
</div>
```

- XMLHttpRequest contains a **responseXML** property.
- responseText is a string, responseXML is a tree.
- Below is a generic way to process XML data with ajax.

```
new Ajax.Request(
    "someURL",
    {
        method : "get",
        onSuccess : someFunction,
    }
);

function someFunction(ajax){
    //do something with ajax.responseXML
}
```

- The responseXML property represents the overall XML document that was sent back.
- We will almost never navigate the tree directly, rather we will use methods to find nodes that match a tag or class.
- **getElementsByTagName(tag)** - returns an array of elements inside the current element that match the parameter.
- **getAttribute(attr)** - returns the value of the supplied node's attribute.

Common Node Properties

- **tagName, nodeName** - the node's tag.
- **nodeValue** - text directly inside the node.
- **nodeType** - integer representing the kind of node, e.g. element, attribute, text, etc. Full list found at http://www.w3schools.com/dom/dom_nodetype.asp
- **attributes** - array of nodes representing the attributes.
- **childNodes** - array of child nodes.
- Other tree properties include: firstChild, lastChild, nextSibling, previousSibling and parentNode.

Gotchas

There are a few things to be aware of with respect to the XML DOM.

- The DOM tree contains nodes for any whitespace or text between elements.
- You cannot access an element's attributes directly by name. Must use the **getAttribute** method.
- Draw out the XML DOM tree for the XML document located at webdev.cs.uwosh.edu/students/krohns/CodeExamples/Ajax/data.xml

- The best way to avoid whitespace nodes is to always fetch groups of nodes by calling getElementsByTagName on some parent node. Also avoid relative links like XXXChild and XXXSibling.
- You cannot access an element's attributes directly by name. Must use the **getAttribute** method.
- Draw out the XML DOM tree for the XML document located at webdev.cs.uwosh.edu/students/krohns/CodeExamples/Ajax/data.xml

Example

- Let's say we have a web page setup to pick a school. We would then use ajax to list all professors who went to that school.
- The partial web page would look something like:

```
<div>
    <p>Universities:</p>
    <ul id="university">
        <li>Retrieving</li>
    </ul>
</div>
<div>
    <p>Professors:</p>
    <ul id="professors">
        <li>Select a university</li>
    </ul>
</div>
```

Example

- Our JavaScript file would start off looking like this.
- We make a call to a php file and have an xml file returned. Code to do this is at:

webdev.cs.uwosh.edu/students/krohns/CodeExamples/Ajax/data.php.txt

```
window.onload = function(){
    new Ajax.Request(
        "data.php",
        {
            method: "GET",
            onSuccess: showUniversities,
        }
    );
}
```

The showUniversities function would look something like this:

```
function showUniversities(ajax){
    var universities = ajax.responseXML.getElementsByTagName
        ("school");
    $("university").removeChild($("university").firstChild.
        nextSibling);
    for(var i=0; i<universities.length; i++){
        var name = universities[i].firstChild.nodeValue;
        var li = document.createElement("li");
        li.innerHTML = name;
        $("university").appendChild(li);
    }
}
```

The universities are now displayed in our list but nothing happens when we click on them. To fix this, we need an event handler attached to each list item.

```
function showUniversities(ajax){
    var universities = ajax.responseXML.getElementsByTagName
        ("school");
    $("university").removeChild($("university").firstChild.
        nextSibling);
    for(var i=0; i<universities.length; i++){
        var name = universities[i].firstChild.nodeValue;
        alert("name: "+name);
        var li = document.createElement("li");
        li.innerHTML = name;
        li.onclick = uniClick;
        $("university").appendChild(li);
    }
}
```

Here we implement the click event handler.

```
function uniClick(ajax){
    new Ajax.Request(
        "data.php",
        {
            method: "GET",
            parameters: {uni: this.innerHTML},
            onSuccess: showPeople
        }
    );
}
```

Finally, the showPeople method is implemented below:

```
function showPeople(ajax){
    //remove old people
    while($("professors").firstChild){
        $("professors").removeChild($("professors").
            firstChild);
    }
    var people = ajax.responseXML.getElementsByTagName("
        person");
    for(var i = 0; i<people.length; i++){
     //get the phone number and name
        var phone = people[i].getAttribute("phone");
        var fullName = people[i].firstChild.nodeValue;
        var li = document.createElement("li");
        li.innerHTML = fullName + " can be reached at " +
            phone;
        $("professors").appendChild(li);
    }
}
```

- Douglas Crockford created an alternative to XML called JavaScript Object Notation or **JSON**.
- JSON is in plain-text format and is readable.
- JSON allows hierarchical data and values within values.
- JSON is generally shorter and less cluttered than XML.

Quick Detour

- In order to appreciate JSON, we need to understand JavaScript objects.
- Code below shows how to create and access instance variables inside an object.

```
var myRect = {
    xPos: 100,
    yPos: 200,
    color: "Red",
    overlaps: ["recOne", "recThree", "recSeven"],
    visible: true,
    myFn: function() { return "Located at (" + this.xPos + "
        ," + this.yPos + ")."}
}

alert(myRect.xPos); //100
alert(myRect["yPos"]); //200
alert(myRect.overlaps[1]); //recThree
alert(myRect.myFn()); //Located at (100, 200).
```

- Consider the following XML data:

```
<people>
    <person school="Iowa" phone="7080">Erik Krohn</person>
    <person school="Iowa" phone="2069">George Thomas</person>
</people>
```

- and the mostly equivalent JSON data:

```
{
  "people": [
    { "school": "Iowa", "phone": "7080", "body": "Erik Krohn" },
    { "school": "Iowa", "phone": "2069", "body": "George Thomas" }
  ]
}
```

- All JSON object properties are enclosed in quotes.
- JSON objects may not have functions as properties.
- Some characters are forbidden in JSON data.
- The main reason to use JSON is that you don't need to traverse or deal with a DOM tree. All properties and data are directly accessible.

- Retrieving JSON data is identical to retrieving XML data.
- The key difference is how you process the data once you receive it.
- JSON data is stored in ajax.responseText.

```
function makeRequest(){
    new Ajax.Request(
        "someURL",
        {
            method: "GET",
            onSuccess: someFunction
        }
    );
}
```

- Using the JSON data from the previous slide.

```
function someFunction(ajax){
    var data = JSON.parse(ajax.responseText);
    alert(data.people[0].phone); //7080
    alert(data.people[1].body); //George Thomas
    alert(data.length); //2
    for(var i=0; i<data.length; i++){
        alert(data.people[i].body + "'s phone number is " +
            data.people[i].phone + " and went to " + data.
            people[i].school + ".");
    }
}
```