

September 13, 2006

Paul Sobocinski (</pub/au/287>)

An Ajax RSS Parser

Ajax (Asynchronous JavaScript And XML) and RSS (Really Simple Syndication) are two technologies that have taken the Web by storm. Most commonly, RSS is used to provide news to either people or other organizations. This is done by serving an "RSS feed" from a website. An RSS feed is simply a link to an XML file that is structured in a certain way. The RSS specification tells us the expected structure of the XML file. For example, the title, author, and description tags are required, and so all RSS XML files will have at least these three tags.

The RSS specification that we will be using is 2.0, which is both the newest and most widely used of the three specifications (0.98, 1.0, and 2.0). Fortunately, RSS 2.0 is far less complex than RSS 1.0, so you can quickly familiarize yourself with RSS 2.0 here: blogs.law.harvard.edu/tech/rss (<http://blogs.law.harvard.edu/tech/rss>). If you want a comprehensive introduction to RSS, covering all three specifications, go here: www.xml.com/pub/a/2002/12/18/dive-into-xml.html (<http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html>).

Why are we using Ajax to parse our RSS? By using Ajax, we are passing over the work of processing the RSS XML file to the web browser, thus reducing server load. Also, Ajax allows the user to have a more seamless web experience, because we are able to fetch the entire RSS XML file from the server without having to refresh the page. Lastly, Ajax is designed to handle XML files, so it's able to parse RSS in a simple and elegant way.

For the purposes of this article, you don't need to be familiar with Ajax; however, a basic understanding of JavaScript is strongly recommended.

Here's how the parser is going to work: first, the file name of the RSS feed is selected in an HTML form. Once the user clicks Submit, the `getRSS()` function is called. This function is responsible for fetching the specified RSS XML file from the server. Once it's fetched successfully, `processRSS()` converts the received XML file into a JavaScript object. Finally, `showRSS(RSS)` is called, which displays some of the information contained in the RSS JavaScript object to us by updating the HTML page. The diagram below summarizes these steps:

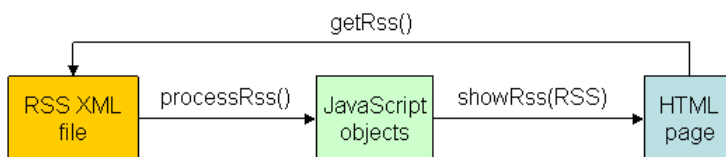


Figure 1. General design

The HTML File

To begin, we'll have a look at the HTML file. The top half (the `form` element) determines which RSS feed to fetch, while the bottom half (the root `div` element) is used to display the information contained in the RSS JavaScript object.

```

<html>

<head>

    <!--B-->

    <script language="javascript" src="rssajax6.js"></script>

    <!--C-->

    <style type="text/css">

        #chan_items { margin: 20px; }

        #chan_items #item { margin-bottom: 10px; }

        #chan_items #item #item_title { font-weight: bold; }

    </style>

</head>

<body>

    <!--A-->

    <form name="rssform" onsubmit="getRSS(); return false;">

        <select name="rssurl">

            <option value="test-rss.xml">test RSS feed</option>

            <option value="google-rss.xml">google RSS feed</option>

        </select>

        <input type="submit" value="fetch rss feed" />

    </form>

    <div id="chan">

        <div id="chan_title"></div>

        <div id="chan_link"></div>

        <div id="chan_description"></div>

        <a id="chan_image_link" href=""></a>

        <div id="chan_items"></div>

        <div id="chan_pubDate"></div>

```

```
<div id="chan_copyright"></div>
```

```
</div>
```

```
</body>
```

```
</html>
```

For now, we will ignore most of the HTML and focus on the form element (labeled `<!--A-->` above). The names of the RSS XML files are specified in the `value` attributes of the option tags of the select element. The user selects one of these files, and then submits the form. The JavaScript that starts the whole process is found in the `onsubmit` tag. After calling the JavaScript function, we add `return false` to prevent the entire form from being sent to the server the "conventional" way. If we'd omitted `return false`, the entire page would refresh and we'd lose all the data that was fetched via Ajax. One last thing: note that the JavaScript code is included in the header as a reference to a separate file (labeled `<!--B-->`). In case you're wondering, the contents of the `<style>` tag (labeled `<!--C-->`) tell the browser how to display the RSS data when it's written to the HTML page by the `showRSS(RSS)` function.

Fetching the XML from the Server: The `getRSS()` Function

The `getRSS()` function is shown below:

```

function getRSS()

{

    /*A*/

    if (window.ActiveXObject) //IE

        xhr = new ActiveXObject("Microsoft.XMLHTTP");

    else if (window.XMLHttpRequest) //other

        xhr = new XMLHttpRequest();

    else

        alert("your browser does not support AJAX");


    /*B*/

    xhr.open("GET",document.rssform.rssurl.value,true);


    /*C*/

    xhr.setRequestHeader("Cache-Control", "no-cache");

    xhr.setRequestHeader("Pragma", "no-cache");


    /*D*/

    xhr.onreadystatechange = function() {

        if (xhr.readyState == 4)

        {

            if (xhr.status == 200)

            {

                /*F*/

                if (xhr.responseText != null)

                    processRSS(xhr.responseXML);

            else

            {

```

```

        alert("Failed to receive RSS file from the server - file not found.");

        return false;
    }

}

else

    alert("Error code " + xhr.status + " received: " + xhr.statusText);

}

}

/*E*/

xhr.send(null);

}

```

Let's walk through this function step by step and figure out what it's doing. The labels in the code (e.g. `/*A*/`) refer to the corresponding explanations below.

A: In order to communicate with the server, we must first define an `XMLHttpRequest` object (XHR). This object is what allows us to connect to the server without refreshing the browser; it is the core of any Ajax application. As we'd expect, Internet Explorer defines the XHR object differently than other browsers, such as Firefox and Safari. We use object detection to determine what browser the JavaScript is running in, and thus define the XHR object appropriately.

B: We set up the XHR object by calling `xhr.open()`. This function takes three arguments: the first is the method we use to fetch the file from the server, the second is the name of the file we are fetching, and the third is set to `true` if we want the response to be received asynchronously. As we are not going to send any request data to the server, a `GET` request is sufficient. The name of the RSS XML file is taken directly from the HTML form. Lastly, we specify an asynchronous response so that we don't have to "wait" for the file to be received—we know when it is available by defining a function that is called when the receiving is complete (more on this later).

C: It is important to request a fresh (non-cached) copy from the server. Here, we set the request header to ensure that this will be the case (`Pragma` is included for backward compatibility).

D: As we have set up an asynchronous connection with the server, the XHR object needs to know what function to call when the server response becomes available. This is the purpose of the `onreadystatechange` property of the XHR object—we set it equal to the function that we want to run when the `readyState` property of the XHR object changes. For our purposes, we only need to be concerned with the `readyState` of `4`, because this indicates that the response is available. We know that we received a successful response if `xhr.status` is equal to `200`. Any other status code means that we didn't receive the response properly.

E: We have set up the XHR object for receiving a response from the server. Now, we call `xhr.send()` to run the server request/response process. As we don't have any data to send, we pass an argument of `null`.

F: If we have reached this point, we know that a response from the server was successfully received, and we're ready to process the XML file. We just do one last check to make sure we didn't get an empty response. We do so by inspecting the `responseText` property (this contains the received file in textual format). Now we're ready to call the `processRSS()` function.

Parsing the XML: The `processRSS()` Function and the `RSS2Channel` Object

The `processRSS()` function is shown below:

```
function processRSS(rssxml)

{

    RSS = new RSS2Channel(rssxml);

    showRSS(RSS);

}
```

This function simply calls the constructor of the `RSS2Channel` object and passes `rssxml`. This argument is of special significance, as it contains all of the RSS information. Moreover, JavaScript is able to recognize this as an XML object, and therefore we are able to use JavaScript's built-in DOM (Document Object Model) functions and properties on it. We can do this because we used the `responseXML` attribute of the XHR object to get the server response. If we had used `responseText`, parsing the XML would be much more difficult.

Now we'll examine the `RSS2Channel` object. Each RSS XML file always has exactly one channel element--this element contains all of the RSS data. As you would expect, this data is organized into a number of sub-elements, or "child" elements. Therefore, `channel` is the root element of an RSS XML file, which is represented by the `RSS2Channel` object. This object is shown below:

```

function RSS2Channel(rssxml)

{

    /*A*/

    /*required string properties*/

    this.title;

    this.link;

    this.description;


    /*optional string properties*/

    this.language;

    this.copyright;

    this.managingEditor;

    this.webMaster;

    this.pubDate;

    this.lastBuildDate;

    this.generator;

    this.docs;

    this.ttl;

    this.rating;


    /*optional object properties*/

    this.category;

    this.image;


    /*array of RSS2Item objects*/

    this.items = new Array();


    /*B*/

```

```

var chanElement = rssxml.getElementsByTagName("channel")[0];

var itemElements = rssxml.getElementsByTagName("item");

/*C*/

for (var i=0; i<itemElements.length; i++)

{

    Item = new RSS2Item(itemElements[i]);

    this.items.push(Item);

}

/*D*/

var properties = new Array("title", "link", "description", "language", "copyright", "managingEditor", "webMa

var tmpElement = null;

for (var i=0; i<properties.length; i++)

{

    tmpElement = chanElement.getElementsByTagName(properties[i])[0];

    if (tmpElement!= null)

        eval("this."+properties[i]+"=tmpElement.childNodes[0].nodeValue");

}

/*E*/

this.category = new RSS2Category(chanElement.getElementsByTagName("category")[0]);

this.image = new RSS2Image(chanElement.getElementsByTagName("image")[0]);

```

As before, we will break the code into smaller pieces and explain each one individually.

A: As a guide, we list out all of the properties that we will be assigning values to. Each of these properties corresponds to an RSS XML element. For example, we will set `this.language` equal to the string found inside the `<language>en-us</language>` XML tag--in this case, `en-us` . Some of these properties will be custom objects, just as `RSS2Channel` . This will be explained in more detail shortly.

B: Here, we create two variables—one to store the contents of the `channel` element, and another to store an array of `item` elements. To accomplish this, we use the `getElementsByTagName()` function, which returns an array of all the elements in the XML file that match a specified tag name. As previously discussed, an RSS XML file only has one `channel` tag, so we expect an array with one element to be returned. We add `[0]` to the end of the function call to get the object and assign it to `chanElement`. On the other hand, we need `itemElements` to be an array, because an RSS XML file will have multiple `<item>` tags.

C: This loop traverses the `itemElements` array and parses each item element individually. An `<item>` tag in an RSS XML file contains a number of child tags, so we need to construct an `RSS2Item` object that will store this data in a meaningful way. We pass the current item element to the constructor, and assign the constructed object to `Item`. This is then added to the `this.items` array. Once this loop is complete, the `items` property of the `RSS2Channel` object will contain an array of custom `RSS2Item` objects. We will talk about the `RSS2Item` object once we're done with `RSS2Channel`.

Use of the `eval()` Function

Before I continue, I wanted to briefly explain the `eval()` function, in case you're unfamiliar with it. This function takes a single argument, which is a string containing the JavaScript code that you want your program to run. For example, `eval('return true')` is identical to `return true`. As you will see, this function is useful when dealing with objects that have a large number of properties.

D: We will now set all of the object properties that take simple strings as their values. As all of these properties are grabbed from the `chanElement` object in the same way, we define an array containing the names of all the properties we want to set, and traverse the array using a `for` loop. To get the actual string value of the XML tag we are examining, we access two properties: `childNodes` and `nodeValue`. The first property exposes all of the child XML elements in the form of an array of objects, while the second property gets the actual string value of the XML element. In the case of the properties being retrieved here, they do not contain any child XML tags, so only one element is returned by `childNodes`. Then, `nodeValue` gets the string value of the element in `childNodes[0]`.

E: Finally, we set the `this.category` and `this.image` properties. Unlike the properties discussed in D, these do have child tags, so we have to construct custom objects for these XML elements (`RSS2Category` and `RSS2Image`, respectively). Let's have a look at the `RSS2Category` function to start:

```
function RSS2Category(catElement)

{

    if (catElement == null) {

        this.domain = null;

        this.value = null;

    } else {

        this.domain = catElement.getAttribute("domain");

        this.value = catElement.childNodes[0].nodeValue;

    }

}
```

This is a simple object with two properties: `domain` and `value`. The `value` property contains the actual contents of the XML tag, while the `domain` property is set to the contents of the XML `domain` tag attribute. For example, a typical `category` XML element looks like this: `<category domain="Syndic8">1765</category>`. In this case, `this.domain` is set to `Syndic8` and `this.value` is set to `1765`. In order to get the `domain` attribute from the XML tag, we use the function `getAttribute()` and pass the tag attribute we want to fetch as a parameter (in this case, `domain`).

As the `image` tag in an RSS XML file has only attributes, the `RSS2Image` constructor makes use of the `getAttribute()` function extensively.

```

function RSS2Image(imgElement)

{

    if (imgElement == null) {

        this.url = null;

        this.link = null;

        this.width = null;

        this.height = null;

        this.description = null;

    } else {

        imgAttribs = new Array("url","title","link","width","height","description");

        for (var i=0; i<imgAttribs.length; i++)

            if (imgElement.getAttribute(imgAttribs[i]) != null)

                eval("this."+imgAttribs[i]+"=imgElement.getAttribute(\""+imgAttribs[i]+"");");

    }

}

```

Now we'll go onto the last remaining property in the `RSS2Channel` object: `items`, which contains an array of `RSS2Item` objects. The code for this object is shown below:

```

function RSS2Item(itemxml)

{

    /*A*/

    /*required properties (strings)*/

    this.title;

    this.link;

    this.description;


    /*optional properties (strings)*/

    this.author;

    this.comments;

    this.pubDate;


    /*optional properties (objects)*/

    this.category;

    this.enclosure;

    this.guid;

    this.source;


    /*B*/

    var properties = new Array("title", "link", "description", "author", "comments", "pubDate");

    var tmpElement = null;

    for (var i=0; i<properties.length; i++)

    {

        tmpElement = itemxml.getElementsByTagName(properties[i])[0];

        if (tmpElement != null)

            eval("this."+properties[i]+"=tmpElement.childNodes[0].nodeValue");

    }
}

```

```

/*C*/

this.category = new RSS2Category(itemxml.getElementsByTagName("category")[0]);

this.enclosure = new RSS2Enclosure(itemxml.getElementsByTagName("enclosure")[0]);

this.guid = new RSS2Guid(itemxml.getElementsByTagName("guid")[0]);

this.source = new RSS2Source(itemxml.getElementsByTagName("source")[0]);

}

```

The `RSS2Item` object is similar to `RSS2Channel` in many ways. We start by listing out the properties that we will be retrieving (A). We then loop through the string properties, and assign each to the contents of its associated XML tag (B). Lastly, we set object properties by calling the appropriate custom object constructor—in each case, passing the XML element that contains the relevant data (C).

The custom objects that are found in the `RSS2Item` object are listed below. They are similar to the `RSS2Category` and `RSS2Image` objects, and they don't use any functions or properties that haven't been discussed earlier.

```

function RSS2Enclosure(encElement)
{
    if (encElement == null) {

        this.url = null;

        this.length = null;

        this.type = null;

    } else {

        this.url = encElement.getAttribute("url");

        this.length = encElement.getAttribute("length");

        this.type = encElement.getAttribute("type");

    }

}

```

```

function RSS2Guid(guidElement)
{
    if (guidElement == null) {

        this.isPermaLink = null;

        this.value = null;

    } else {

        this.isPermaLink = guidElement.getAttribute("isPermaLink");

        this.value = guidElement.childNodes[0].nodeValue;

    }

}

```

```

function RSS2Source(souElement)
{
    if (souElement == null) {

        this.url = null;

    }
}

```

```

        this.value = null;

    } else {

        this.url = souElement.getAttribute("url");

        this.value = souElement.childNodes[0].nodeValue;

    }

}

```

Now that we've fully defined our RSS object, we can move on to the last step: displaying its actual content.

Displaying the RSS Data: The showRSS(RSS) Function

Before we go into the JavaScript code for the showRSS(RSS) function, let's have a look at the root div element of the HTML page mentioned earlier:

```

<div class="rss" id="chan">

    <div class="rss" id="chan_title"></div>

    <div class="rss" id="chan_link"></div>

    <div class="rss" id="chan_description"></div>

    <a class="rss" id="chan_image_link" href=""></a>

    <div class="rss" id="chan_items"></div>

    <div class="rss" id="chan_pubDate"></div>

    <div class="rss" id="chan_copyright"></div>

</div>

```

As you can see, the root div element has a number of child div tags. These tags will be populated with the data in the RSS object by the showRSS(RSS) function, which is shown below.

```

function showRSS(RSS)

{

    /*A*/

    var imageTag = "<img id='chan_image'";

    var startItemTag = "<div id='item'>";

    var startTitle = "<div id='item_title'>";

    var startLink = "<div id='item_link'>";

    var startDescription = "<div id='item_description'>";

    var endTag = "</div>";


    /*B*/

    var properties = new Array("title","link","description","pubDate","copyright");

    for (var i=0; i<properties.length; i++)

    {

        eval("document.getElementById('chan_'+properties[i]+'').innerHTML = ''); /*B1*/

        curProp = eval("RSS."+properties[i]);

        if (curProp != null)

            eval("document.getElementById('chan_'+properties[i]+'').innerHTML = curProp"); /*B2*/

    }


    /*C*/

    /*show the image*/

    document.getElementById("chan_image_link").innerHTML = "";

    if (RSS.image.src != null)

    {

        document.getElementById("chan_image_link").href = RSS.image.link; /*C1*/

        document.getElementById("chan_image_link").innerHTML = imageTag

            +" alt='"+RSS.image.description

```

```

        +'' width='"+RSS.image.width

        +'' height='"+RSS.image.height

        +'' src='"+RSS.image.url

        +'' ">"; /*C2*/

    }

/*D*/

document.getElementById("chan_items").innerHTML = "";

for (var i=0; i<RSS.items.length; i++)

{

    item_html = startItemTag;

    item_html += (RSS.items[i].title == null) ? "" : startTitle + RSS.items[i].title + endTag;

    item_html += (RSS.items[i].link == null) ? "" : startLink + RSS.items[i].link + endTag;

    item_html += (RSS.items[i].description == null) ? "" : startDescription + RSS.items[i].description + end

    item_html += endTag;

    document.getElementById("chan_items").innerHTML += item_html; /*D1*/

}

return true;

```

A: As we have no way of knowing the number of channel items in the RSS feed, we must dynamically generate the HTML for the RSS items. These are the default values for the HTML tags that will contain the `RSS2Item` data. For compatibility, we also dynamically generate the `img` HTML tag.

B: We traverse the string properties in the `RSS2Category` object here, similar to how we did in the constructor. In order to clear any data that may remain from an old RSS feed, we reset the `innerHTML` property on line B1. We are able to fetch the specific `div` element that we need from the HTML by calling `getElementById()`. Providing that the property is defined, we set the `div` element to its new value on line B2.

C: Again, we use the `getElementById()` function to get the HTML element that will contain the image from the RSS feed. As the image should be linkable, we use an anchor element (`a`) instead of a `div` element. The `href` attribute in the anchor element specifies what the image should link to, so we assign it to the value found in `RSS.image.link` (C1). The content of the element is filled in using the `innerHTML` property, as we have done in part B (C2).

D: Here is where we display the items in the RSS object. A `div` tag is defined for each RSS item, containing the title, link, and description. For the sake of clarity, the other properties have been omitted. Each `div` tag is appended to the contents of the `chan_items` parent tag using the `innerHTML` property (D1).

Wrap-Up

The Ajax RSS parser has been tested in IE 6.0 and Firefox 1.5.0.6 for Windows XP. The `RSS2Channel` object does not support all of the elements in the RSS 2.0 specification. The ones that have been omitted are `cloud`, `textInput`, `skipHours`, and `skipDays`. For the most part, these RSS elements are only useful on the server side, so it wouldn't make sense to include them in a client-side parser.

After noting the length of the code, you may be thinking that the same functionality could have been accomplished with half the number of lines of code. In particular, we could have completely omitted the RSS object by writing the `showRSS(RSS)` function in a way that reads the RSS properties directly from the XML element. Certainly, this is possible. However, `showRSS()` is only meant to be an example of how the `RSS2Channel` object can be used. By defining an RSS object that contains meaningful RSS data, we have a much more scalable application. For example, the code can be easily extended to fetch multiple feeds. The RSS objects from these feeds can then be manipulated, or compared with other feeds (you can fetch a new feed after a certain interval, and compare it with the old one). The point of a separate RSS object is to make increasingly complex applications like this easier to develop.

All of the files that were discussed are available below:

The HTML file: [rssajax.html \(/pub/a/2006/09/13/examples/rssajax.html\)](/pub/a/2006/09/13/examples/rssajax.html).

The JavaScript file, containing the RSS parser: [rssajax.js \(/pub/a/2006/09/13/examples/rssajax.js\)](/pub/a/2006/09/13/examples/rssajax.js).

Sample RSS file 1: [test-rss.xml \(/pub/a/2006/09/13/examples/test-rss.xml\)](/pub/a/2006/09/13/examples/test-rss.xml)

Sample RSS file 2: [google-rss.xml \(/pub/a/2006/09/13/examples/google-rss.xml\)](/pub/a/2006/09/13/examples/google-rss.xml)

Content licensed from and © 1998 - 2008 O'Reilly Media, Inc.