

AJAX Interfaces

Table of contents

1 Introduction.....	2
2 AJAX Intro.....	2
3 QDF Interfaces.....	2
3.1 Choose Your Weapon.....	2
3.2 Interface Legos.....	3
3.3 Loading The Interface.....	4
3.4 Coordinating Things.....	5
4 Conclusion.....	6

1. Introduction

This article will be much less formal than Article 1 or Article 2. I'll be discussing how I use [AJAX](http://en.wikipedia.org/wiki/AJAX) (<http://en.wikipedia.org/wiki/AJAX>) to create highly-interactive web interfaces. I'll mainly be focusing on implementing these interfaces in straight (X)HTML but it should be fairly analagous for use in portlets and JSP pages.

Article 1 described one technique, *the DIV swap*, for creating interactive interfaces that don't require a page refresh for every action performed by users. This technique works well when you know the all of the content a priori. Some examples include slideshow-esque applications or if you have a bit of extra info that you want to show the user at their request. The problem is, the technique doesn't scale very well when there are a large number of hidden elements. The more hidden elements, the longer the download time (since the user has to download everything) and the longer the update time (because you are visiting many elements).

To combat the limitations of the DIV swap, I decided to explore how I could improve the interfaces with AJAX.

2. AJAX Intro

If you are unfamiliar with AJAX, check out the [Wikipedia article](http://en.wikipedia.org/wiki/AJAX) (<http://en.wikipedia.org/wiki/AJAX>) on it. This provides a good introduction to concepts. As with any technology, there are multiple ways to skin the cat so there are multiple AJAX libraries. Each has its pros and cons, so you'll have to evaluate them to figure out which meets your needs best. I've linked a few below (the list is by no means exhaustive or in any particular order):

- [DWR](http://www.getahead.ltd.uk/dwr/index.html) (<http://www.getahead.ltd.uk/dwr/index.html>) - Use Javascript/AJAX to call methods in Java code server side.
- [Dojo](http://dojotoolkit.org/) (<http://dojotoolkit.org/>) - solid AJAX functionality with work in many areas including widgets, animation, etc.
- [Prototype](http://prototype.conio.net/) (<http://prototype.conio.net/>) - Big in the Ruby on Rails community; AJAX functionality as well as some other useful features.
- [Sarissa](http://sarissa.sourceforge.net/doc/) (<http://sarissa.sourceforge.net/doc/>) - AJAX functionality with a focus on XML.

3. QDF Interfaces

- [AJAXified QDF Interfaces](http://cws3.geol.iastate.edu:8080/xqe/) (<http://cws3.geol.iastate.edu:8080/xqe/>)

3.1. Choose Your Weapon

AJAX Interfaces

For the QDF interfaces, I needed to composite content from multiple locations into a single, interactive interface. Since I wasn't calling any Java code, I ruled out DWR. And because my content was going to be mixed HTML, text, images, I ruled out Sarissa. Ultimately I decided on Dojo. I used Prototype first, with much success, but I encountered some compatibility issues that I didn't have with Dojo.

Note:

I didn't have time to track down the exact problem and contact the Prototype group with a bug report. The problem may be fixed in future releases (I was using 1.2.0) but since Dojo just worked for me, I'm sticking with it.

3.2. Interface Legos

The general approach I took to developing the new interfaces was to divide things up into various content 'Lego' blocks. Then I could assemble the full interface from these smaller blocks of content. This approach also allowed me to reuse the various content blocks elsewhere and also replace certain blocks dynamically without disturbing the rest of the interface.

To illustrate this, view the source of the **AJAXified QDF Interfaces** link above. You will notice that the page is devoid of any of the content shown on the screen. It merely mocks up and arranges where the various content Legos will go. The magic happens in the `initUI()` function called by `body.onload`. Let's look a little more closely at this function:

```
/*
 * Tests whether the interface is possible. If it is, it proceeds to
 * load things.
 * Otherwise, displays an error message to the user.
 */
function initUI() {
    // load scripts/test.js which contains 'true'
    dojo.io.bind({
        url: "scripts/test.js",
        load: function(type, evaldObj) { if (evaldObj) loadUI(); else
bailOut(); },
        error: function(type, error) { bailOut(); },
        mimetype: "text/javascript"
    });
}
```

Here we catch our first glimpse of Dojo in action. Loading remote content is accomplished via the `dojo.io.bind()` function. As you can see from the snippet above, it takes a few parameters. Most should be self explanatory, but check out this [article](http://dojotoolkit.org/intro_to_dojo_io.html) (http://dojotoolkit.org/intro_to_dojo_io.html) for more information.

This function just does an initial test to make sure the browser supports the XMLHttpRequest

functionality. It tries to load a simple javascript [script](#) (<http://cws3.geol.iastate.edu:8080/xqe/scripts/test.js>) . If it succeeds, then we call the `loadUI()` function to set up the interface. Otherwise, it writes out an error message to the user.

3.3. Loading The Interface

Assuming things worked, the `loadUI()` function is called. This is where all of the various Lego blocks are assembled into a cohesive interface. The code is below:

```

/*
 *   Dynamically loads the interface.
 */
function loadUI() {
    // initialize our UI containers
    oMenu = getElementById('menu');
    oMain = getElementById('main');
    oHelp = getElementById('help');
    oSearchInfo = getElementById('searchinfo');
    oQueryInfo = getElementById('queryinfo');

    // load the menu contents
    loadRemoteContent(oMenu, baseUrl +
'qdf?target=all&serializeAs=xqi-menu', true);

    // get the user's last query from the cookie
    var lastQuery = getCookie('last-query');

    // if the last query wasn't set or was 'splash' show our splash page
    if (lastQuery == null || lastQuery == '')
        showStatic('splash');
    else
        showStatic(lastQuery);

    // load in the quick links
    loadRemoteContent(oHelp, 'help.html', true);

    // hide the results
    clearResults();
}

```

The main function of interest is `loadRemoteContent()`. This is basically a convenient wrapper around `dojo.io.bind` that takes an element and updates its contents with data retrieved remotely. The function does a few convenient things:

1. Puts a nice 'Loading...' message into the container
2. Fires off an asynchronous request for the remote content
3. If you specify 'true' as the last argument, it looks through the returned content and executes and `<script>` tags.

AJAX Interfaces

4. It writes the content into the container
5. In the event of an error, it writes a nice error message into the container

The code for the method is below. Step #3 is interesting because the `<script>` tags get executed after the new content is loaded, so remote content can trigger changes and events in the interface when they are loaded. This is extremely powerful.

```
/*
 * Loads content from a URL and puts it into a container element. If an
 * error occurs, the loadError message is put into the
 * container instead. You can also customize the messages by creating
 * two variables called loadMsg and loadError. If you
 * pass true for execute, inline scripts in the remote content will be
 * executed.
 *
 * Example:
 * loadRemoteContent(getElementById('some-div'), 'content.html', true);
 */
function loadRemoteContent(container, url, execute) {
    loadContent(container, ((loadMsg) ? loadMsg : ''));
    try {
        dojo.io.bind({
            url: url,
            load: function(type, data, evt) { loadContent(container,
                ((execute) ? executeScripts(data) : data)); },
            error: function(type, error) { loadContent(container,
                ((loadError) ? loadError : '')); },
            mimetype: "text/plain"
        });
    } catch (e) {
        loadContent(container, ((loadError) ? loadError : ''));
    }
}
```

Note:

There is one mildly annoying caveat with using the XMLHttpRequest object: you can't reliably load content from outside the current domain. This is due to browser security policies. They can be turned off but it has to be done manually.

The code references a two other functions (`loadContent()` and `executeScripts()`). Both of these are fairly straightforward. You can find them in [dhtml.js](http://cws3.geol.iastate.edu:8080/xqe/scripts/dhtml.js) (<http://cws3.geol.iastate.edu:8080/xqe/scripts/dhtml.js>) (which contains many other useful methods for doing AJAX and DHTML stuff).

3.4. Coordinating Things

The rest of the magic in the QDF interfaces comes from loading content into specific areas in response to user events. For example, when the user clicks on an entry in the menu, I load the search interface for that particular menu entry into the center column. The interface is

generated via an XSLT of the QDF document as covered in Articles 1 and 2. Likewise, when the user fills in the form and submits a query, I build a URL to the results and load the content into the designated 'results' area.

The right column is something I'm proud of. It displays context-sensitive information. When you first visit, there is just some generic information in a box on the right. When you click on a search, you get a little more information about the search you clicked on. When you actually submit a search, you get a bit more information. The goal is to develop this feature more fully so that new information is popping up in the right column as it becomes available. All of this is still accomplished the same way as the rest of it. I have divided the right column into 3 sections. Each gets some content depending on which stage the user is at.

There are a few other nice features hidden away. One is dynamically adjusting the menu height based on the size of the window. For example, if you load up the interface, the height of the menu is adjusted so the 'Show Search Groups' box is visible. It also re-adjusts the height when you switch groups. I also do some tricks with cookies to persist interface state (nothing revolutionary, but it is the little things that count). So if you navigate away from the interface and then come back, you'll be at the search you were looking at prior to leaving. Finally, changing the colors of the boxes in the right column based on 'priority' is also a nice touch.

4. Conclusion

Hopefully this gave you an idea of some of the things that can be accomplished via AJAX to create highly interactive interfaces. Obviously what I've done is only a small subset of what can be accomplished with AJAX. For more examples, check out [this](http://www.fiftyfoureleven.com/resources/programming/xmlhttprequest/examples) (<http://www.fiftyfoureleven.com/resources/programming/xmlhttprequest/examples>) , [this](http://ajax2go.com/index.php?itemid=17) (<http://ajax2go.com/index.php?itemid=17>) , and [this](http://www.ajaxian.com/) (<http://www.ajaxian.com/>) or just do a search for "AJAX examples".

Feel free to peruse my [scripts directory](http://cws3.geol.iastate.edu:8080/xqe/scripts/) (<http://cws3.geol.iastate.edu:8080/xqe/scripts/>) . And if you have any questions, comments, or suggestions feel free to contact me at jareed@iastate.edu (<mailto:jareed@iastate.edu>) .