The AjaxParts Taglib from Java Web Parts: AJAX for Java Developers the Easy (yet powerful) Way!

Frank W. Zammetti

Introduction

Many technologies and programming techniques come and go. Some linger a little while, and then fade away. Others gestate in the collective consciousness of the developer community for a while until finally they fundamentally change the way people think about developing software. Other times, something comes on the scene, has a major impact, and is with us for a very long time after that.

AJAX is one of those rare things that falls into both of the later two groups.

As you undoubtedly know by now, unless you've been sequestered on a mob trial jury for the past two years, AJAX stands for Asynchronous JavaScript and XML. AJAX is, at its simplest level, a method whereby clients, web pages loaded in a user's browser in this case, can make requests to the server that are atypical in that they do not automatically result in the enter page being regenerated, as is normally the case when a user clicks a link or submits a form or performs some other action that invokes the server. In fact, an AJAX request will have no discernable effect to the user by itself. There needs to be some JavaScript that actually does something with the response, i.e., updates what the user sees.

As the name implies, AJAX requires JavaScript (well, **technically** it doesn't, but for all intents and purposes, to most people, it does… more frequently though it doesn't involve XML at all, and sometimes it isn't even asynchronous!). This requirement of JavaScript has resulted in web developers, who previously may not have done much client-side development, having to learn a whole new realm of programming. Now, to be sure, JavaScript isn't really all that difficult (although it can get complex at higher levels). By and large though, successful Java programmers, or successful programmers of any kind for that matter, tend to be able to pick up JavaScript without too much difficulty.

JavaScript has also historically had a bad reputation and something "script kiddies" use, not real, serious developers. Lately though, this perception has been changing, as developers begin to understand how to write JavaScript that is well-organized, object-oriented and as "enterprise" as what they do on the server.

AJAX also tends to leads to some tricky situations because of its (usually) asynchronous nature. What happens if two requests get fired at around the same time? Should the second one cancel out the first? If they should both be allowed to complete, is the order important? Whether order matters or not, what if they are both modifying the same data structures, or portion of the screen? What about the cross-browser issues, since as I'm sure you know, the XMLHttpRequest object that is the heart of most AJAX functionality isn't instantiated the same on all browsers? How should error handling be done? All of these issues, and more, are things you will have to confront if you are intent on writing your own AJAX code.

For these reasons and more, many people have decided that a good JavaScript/AJAX library is the way to go. And, in the blink of an eye seemingly, a plethora of such libraries have come into existence. Some are specifically geared towards AJAX, some examples of which are Bajax (https://developer.berlios.de/projects/bajax), SACK (http://twilightuniverse.com/projects/sack) and Tibet (http://www.technicalpursuit.com/ajax.htm). Other libraries have a much wider focus, with AJAX playing an important, but not exclusive, role, some examples of which are Dojo (http://dojotoolkit.org), Prototype (http://prototype.conio.net) and MochiKit (http://www.mochikit.com).

Ahem. Oh yes, and Java Web Parts (JWP). More specifically, the AjaxParts Taglib component of JWP (http://javawebparts.sourceforge.net)..

JWP isn't a JavaScript library per se. It is in fact a library of components of interest to Java web developers in general (and not just web developers in some cases). Things like servlets, servlet filters, taglibs, utility classes for working with request, response and session objects, can be found in JWP, and much more. One of the things it provides are some taglibs for generating JavaScript code. One of those taglibs, in fact the component that really began JWP, is the AjaxParts Taglib.

The AjaxParts Taglib (heretofore referred to as APT) is one of those libraries focused exclusively on AJAX. What makes APT different to a large extent from the libraries named above, and nearly every other AJAX library out there, is that it is geared specifically towards Java developers because it is implemented as a custom taglib. This may sound limiting, since only those doing Java web development can use APT, but we prefer to keep the focus narrow to ensure the result is as effective as it can be. What this allows for is AJAX without any coding, without any need to know JavaScript, or how client-side development works.

APT takes a declarative, event-driven approach to AJAX, something else which separates it from the rest (the declarative nature at least; there are other event-driven libraries out there). Using APT you can get into the AJAX game without writing a bit of JavaScript yourself! All it takes is a few new tags in your pages (two that are used all the time, two others not as much) and some XML configuration. In addition, APT makes it child's play to add AJAX functionality to existing web applications without modifying them in any significant way.

APT handles all the sticky cross-browser issues for you, as any AJAX library should do. It also deals with simultaneous requests by assuring that they will always complete and not step on each others' data (even if they were to the same URL and used all the same data, they will still be handled completely independent of one another).

A First Simple Example

Rather than try and convince you of the benefits APT offers, let's just get right to it and have a look at a simple example so you can see for yourself. In this example, we will implement a typical AJAX function: when a button is clicked by the user, a request will be made to the server, and the response, which we'll assume is a snippet of HTML markup, will be inserted into a on the page.

First, let's see what is required to get things set up. We'll assume we have a simple webapp already build. Nothing fancy, just a single index.jsp in the root, and the web.xml file shown in Listing 1.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC  "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <context-param>
    <param-name>AjaxPartsTaglibConfig</param-name>
    <param-value>/WEB-INF/ajax-config.xml</param-value>
  </context-param>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
```

```
</web-app>
```

Listing 1. web.xml for the first example application.

As you can see, we have a single entry in web.xml that deals with APT. The *AjaxPartsTaglibConfig* context parameter tells APT where its configuration files (this can be a comma-separated list of filenames) can be found. This is a context-relative path and filename.

So, what does ajax-config.xml look like? Let's answer that right now, as seen in Listing 2.

```
<ajaxConfig>

  <group ajaxRef="MyFunctions">
    <element ajaxRef="Button1">
      <event type="onclick">
        <requestHandler type="std:SimpleRequest" target="result1.htm">
          <parameter />
        </requestHandler>
        <responseHandler type="std:InnerHTML">
          <parameter>resultDiv</parameter>
        </responseHandler>
      </event>
    </element>
  </group>

</ajaxConfig>
```

Listing 2. The ajax-config.xml file for the first example application.

An APT configuration file always has the root element <ajaxConfig>, and underneath that are as many <group> elements as you wish. A <group> is really just a container; it doesn't necessarily correspond to an HTML form, or a page, JSP, or anything else. In practice though, we recommend a single <group> for each logical view in your web application, which usually corresponds to a JSP. However, the organization you use is entirely up to you.

Also note that in terms of organization, since you can specify multiple config files in the *AjaxPartsTaglibConfig* parameter, you can break things up however you wish. If one monolithic config file suites you, that is no problem. If you would prefer to have one config file per JSP, that is just as viable. The choice is entirely yours.

Within a <group> are as many <element> elements as you wish. An <element> corresponds to an element on your page that will be the source of an AJAX request, or event, as it is referred to in APT parlance. This could be a button, a , a link, or anything else you want. It is also possible to configure an AJAX event that is not attached to a specific page element. These can fire periodically via timer, or can be executed manually by client code. More on these features later!

For each element, you can configure one or more events. Each event is represented by an <event> element. The *type* attribute specifies the event that will fire this request. This can be any of the typical DOM event handlers, such as onclick, onmouseover, onblur, etc.

Under each <event> is two elements, <requestHandler> and <responseHandler>. These are the keys to determining what will happen when this event is fired. The request handler determines what is sent to the server. APT provides a variety of what are called "standard" handlers. These try to cover all the most common AJAX use cases people need. The *type* attribute of the <requestHandler> element names the request handler that will be used. If that value begins with "std:", then it is one of the built-in standard handlers. In the example shown above, we are using the SimpleRequest handler, which simply sends a request to the URI specified by the *target* attribute. No data is passed with this request, so this is suitable for "trigger"-type requests, i.e., when the content being returned isn't dependant on any input.

Note the <parameter> element that is a child of <requestHandler>. This value is specific to the handler specified and can be virtually anything. Sometimes it is a list of form elements to get values from. Sometimes it is a list of DOM IDs to get values from. Sometimes it is the members of an XML document to be constructed and sent to the server. If you write your own custom handler, as is discussed a bit later, this parameter can have whatever meaning you need or wan it to have. Or it might not be used at all, again, your choice entirely.

Next up we see the <responseHandler>. Most of what was said about the request handlers applies here too, although as you would expect, there is no *target* attribute. The <parameter> element again is handler-specific. In this example, we are using the standard InnerHTML handler, which simply takes the response from the server and inserts into the element, using the innerHTML property as you would expect, named by the <parameter> value.

The true benefit to this declarative approach of course is that if you need to change the URI an AJAX request goes to, or want to do something completely different with the response, i.e., maybe you want to pop an alert() instead of updating a later on, there is no coding changes, no recompiling… just modify the config file and your good to go.

Now, let's talk about what the JSP here might look like. It looks like Listing 3.

```
<%@ taglib prefix="ajax" uri="javawebparts/ajaxparts/taglib" %>

<html>
<head>
  <title>APT Example</title>
</head>

<body>

  <input type="button" value="Click me for AJAX"><ajax:event
ajaxRef="MyFunctions/Button1" />
  <br><br>
  <div id="resultDiv"></div>

<ajax:enable />
</body>
</html>
```

Listing 3. index.jsp for the first example app.

As previously mentioned, the beauty of APT is that there is no client-side coding required. Two custom tags provide most of the functionality: <ajax:event> and <ajax:enable>.

<ajax:event> is used to “attach” an AJAX event, as configured in the config file, to a given page element. In this case, we want to attach the event we saw configured above to the button element. Note that the <ajax:event> tag **must** follow the element the event is to be attached to **exactly**. In other words, **you can’t even have spaces between them**! If this rule is not followed, all manner of nastiness will ensue.

How does the configuration data get tied to the element you ask? It’s all based on the required *ajaxRef* attribute value. Take a look at the configuration file again and note the *ajaxRef* values for the <group> element and the <element> element. The value of the *ajaxRef* attribute on the <ajax:event> tag is the combination of the two. The *ajaxRef* of the <group> is “MyFunctions”, and the *ajaxRef* of the <element> is “Button1”. Therefore, to refer to the event(s) configured for that <element>, the *ajaxRef* of the <ajax:event> tag is these two values separate by a forward-slash, or “MyFunctions/Button1”, as see in the JSP code. Also note the implication here: all the events configured for that element will be attached to that page element, there is no way to select which ones are attached (this is an enhancement slated for a future release). In general though, this isn’t a problem, because most of the time an <element> element in the config file should correspond to a physical page element.

Because the combination of the two *ajaxRef* values, one each for the <group> and <element>, form a hierarchy, you can organize your configurations however you wish, and can set it up so you can share configurations on multiple elements (there is nothing that says you can’t use the same configuration for multiple elements, whether on the same page or a different page).

The second step, **after** all <ajax:event> tags on the page is the <ajax:enable> tag. This is where the majority of the work is actually done. This tag is responsible for rendering the JavaScript that is required to make APT work. Because the determination of what code is actually required can only be done after all the <ajax:event> tags have processed, the <ajax:enable>: tag **must** appear after all of them. Therefore, we recommend always putting it immediately before the closing <body> tag. This tag has a couple of attributes that are of interest:

- debug (optional, default: "error")- This sets the level of debugging that will occur on the client-side. The possible values are "trace", "debug", "info", "error", "fatal". Each value, in that order, will result in less debug messages (i.e., the setting "trace" will result in the most verbose logging, and "fatal" will result in the least.

- logger (optional, default: "JWPAlertLogger")- This sets the logger object that will be used to record logging messages. Two loggers are available: "JWPAlertLogger", which just dislays logging messages with a JavaScript alert(), and "JWPWindowLogger" which opens a new window and writes logging messages to it.

- suppress (optional, default: "false")- This determines whether the logger object and main APT object that really is the heart of APT on the client, is rendered. You see, you **can** use APT in a JSP that renders a response to an AJAX request. In that situation however, you do not want the main object and logger objects to be rendered because conflicts would arise if they were. So, setting suppress to “true” allows you to suppress generation of that code and avoid those conflicts.

The only other piece of the puzzle is adding two JARs to your application, javawebparts-ajaxparts-xxx.jar, where xxx is the version, and javawebparts-acore-xxx.jar (the core JAR is required by all Java Web Parts packages). The only dependency outside JWP is Jakarta Commons Logging (JCL)..

If you execute the first sample app that accompanies this article (aptexample1), and click the button, you will see a string of text appear below the button. It has been inserted into a as the result of an AJAX event request, as defined in the configuration file. The result is rendered via an HTML file, just to avoid actually needing a server component. This by the way is a great technique to have in your toolbox… developing AJAX

applications and using “dummy” HTML pages early on in place of a real server saves you from having to work out all the server details and just get the client-side right first.

Other Configuration Options

The example configuration file shown in Listing 2 is very simple and does not use all the capabilities APT offers. To get a feel for those capabilities, let’s look at a more complex configuration file, shown in Listing 4.

```
<ajaxConfig>

  <handler name="alertErrorHandlerElement" type="error">
    <location>local</location>
    <function>alertErrorHandlerElement</function>
  </handler>

  <group ajaxRef="MyFunctions">
    <element ajaxRef="Button1" preProc="testPre" postProc="testPost"
    form="parent" async="true">
      <event type="onclick">
        <errorHandler code="404" type="alertErrorHandlerElement" />
        <requestHandler type="std:QueryString" target="result1.htm">
          <parameter>text1V=text1,text2V=text2</parameter>
        </requestHandler>
        <responseHandler type=”std:InnerHTML” matchPattern="/^r2ok/">
          <parameter>resultDiv</parameter>
        </responseHandler>
        <responseHandler type="std:Alerter">
          <parameter />
        </responseHandler>
      </event>
    </element>
  </group>

</ajaxConfig>
```

Listing 4. A more complex ajax-config.xml file.

Ok, so, let’s examine this step by step. First up is the all-new <handler> element. Recall that there are a number of built-in standard handlers that APT provides out-of-the-box. Well, what happens if they don’t quite meet your needs? This is where you may have to write some JavaScript yourself. APT allows you to define a handler, and use it in your configuration, just as you would a standard handler. The <handler> element defines the *name* of the handler (which will be the value of the *type* attribute of the <requestHandler>, <responseHandler> or <errorHandler> elements). The *type* attribute of the <handler> element tells what kind of handler it is, and the valid values are “request”, “response” and “error”. The <location> child element is one of two things: either the value “local”, which means the developer has full responsibility for ensuring the JavaScript for this handler is present on the page that uses it, or it is a URI that will be used to render a JavaScript link reference to import the code to the page. Finally, the <function> element determines the name of the JavaScript function that will be called.

You probably noticed mention of error handlers there. This is another capability APT provides. You can define a JavaScript function to execute when specified errors occur while processing an AJAX request. To do this, you add an <errorHandler> element, either as a child of a <group> element, <element> element, or <event> element. Note that handlers defined at "lower" level will override those defined at "higher" levels. In other words, if you have an <errorHandler> element as a child of a <group> element, and then have one as a child of an <event> element, the handler that is the child of the <event> element will be used for any errors that occur processing that event. Any <event> elements in the same group that do not define their own <errorHandler> element will use the one defined as a child of the <group> element.

When you define an <errorHandler>, you specify the HTTP response code that the handler should, well, handle, via the *code* attribute. And yes, you can handle 404's differently from 500's if you wish by defining multiple <errorHandler> elements, one for each response code. The *type* attribute, as is probably obvious by name, names the handler that will deal with that particular code.

The next new items we see are some attribute on the <element> element. First up is *preProc*. This names a JavaScript function that will fire before any event fires, and you can do whatever you wish in it. Next is *postProc*, which is essentially the same as *preProc* except that it fires after all response handlers have fired, and after all <script> blocks in the server's response have been evaluated. After that is the *form* attribute. Some of the request handlers, QueryString for instance, requires a reference to an HTML form to generate the data it will send to the server. The *form* attribute allows you to specify the name (or DOM ID) of the form the handler will use. You can also use the special value "parent", which indicates the handler should use the form that is the parent of the element on the page the event is attached to. Lastly, the *async* attribute, which is set to either "true" or "false", indicates whether the AJAX request should be asynchronous or not. This defaults to "true", and chances are you will want to leave it that way most of the time (setting it to "false" means, for all intents and purposes, that the page is locked, i.e., unresponsive to the user or other JavaScript, until the request completes… not typically what you want to happen).

Something else we see here is the ability to have multiple <responseHandler> definitions. You can in fact have as many as you want! They essentially form a chain, and will be executed in the order they appear in the configuration. Like all good chains, they can be broken. If a custom handler is listed, and that handler returns false (handlers generally don't return any value), then the remainder of the chain will be aborted.

Lastly, we see a new attribute on the <responseHandler>, namely *matchPattern*. This attribute is used to determine if the handler should fire or not. This attribute is a regex expression that is used to examine the response from the server. If the response matches this pattern, only then will the response handler fire. Because you can have multiple response handlers per event, this allows you the possibility of only having certain ones execute, depending on the returned response. For instance, in the configuration in Listing 4, the first <responseHandler> will only execute if the response contains the text "ok", but the second <responseHandler> will always fire regardless.

Out-Of-The-Box Power

As mentioned, APT provides a number of standard handlers built-in. These try to cover the major bases developers doing AJAX need. Let's have a look at them, shall we?

Let's first look at the request handlers supplied and see what they do:

- SimpleRequest – As seen already, this sends a "naked" request, that is, one with no parameters or dynamic content, to the server.

- QueryString – This constructs a query string from a form and appends it to the target URI. The <parameter> element allows you to specify the names of the parameters to send, and the source (i.e., form fields) of their values.

- SendById – This constructs a query string (or POST body) based on explicit elements based on DOM ID and sends it to the specified target. The <parameter> element allows you to specify a list of IDs to get values from, and what property to read for that value (i.e., innerHTML, innerText, etc.). In addition, this handler can transmit that data as a query string or as XML (in which case you also have the opportunity to define the root element of the XML document to be constructed).

- SimpleXML – This constructs a simple XML document from a form and sends it as the POST body. The <parameter> element allows you to specify the root element, as well as each of the elements that will appear in the document, and the HTML form element that will be the source for the value of each element.

- Poster – This works just like the QueryString handler, except that it submits its data as a POST body.

Now let's have a look at the response handlers:

- Alerter – Pops an alert(), showing the content returned by the server.

- CodeExecuter – This assumes the response from the server is JavaScript and simply executes it. Note that the content should **not** be wrapped in <script> tags.

- IFrameDisplay – This handler takes the content returned by the server and writes it to a named iFrame on the page.

- InnerHTML – Writes the returned content to an element on the page, usually a (but could be other elements), via the innerHTML property.

- DoNothing – A seemingly pointless handler that does the obvious: nothing! An important fact to realize is that **all** responses received via AJAX requests will be scanned for <script> blocks, and those blocks evaluated, regardless of the handler used. So, what if you only want to execute some JavaScript returned by the server? In that case, you still need to specify a response handler, and that's where this one comes in. The <script> blocks will be evaluated, but the handler will do nothing else. Attentive readers will realize that this sounds very similar to CodeExecuter, but there is in fact a difference. The difference is that CodeExecuter expects the servers' response to be JavaScript, and if its anything else, errors will likely occur. What if you want to include some sort of textual content as well? You might want to include some markup, or other type of data, along with the JavaScript. In that case, CodeExecuter wouldn't work as expected, so using DoNothing in that case would allow the JavaScript to still be executed, even though it is potentially "mixed" content.

- Redirector – This assumes the response from the server is a URL, and redirects the client to it. Alternatively, you can specify the target URL in the config file, in which case the actual response is more or less irrelevant. This is frequently used to handle things like session timeouts… a developer will have a filter that detects when a session has timed out. If it has, it forwards to a JSP that renders a special response string, maybe something as simple as "timeout", and that's all it renders. In the ajax-config.xml file, a <responseHandler> of type Redirector is configured with a specified JSP named that deals with timeouts (i.e., returning to a logon page for example). Then, for all <event>'s defined, this <responseHandler> appears as the first handler in the chain. It examines the response for that specific text, using the *matchPattern* attribute, and if it finds that special string, the handler fires, otherwise it

does not. If it fires, the net result is that the user is redirected to the appropriate "session timeout" page.

- Selectbox - This handler populates a <select> element with data returns by the server. The server process, which you are responsible for writing, must return XML in the following form:

  ```
  <list>
    <option value="??">????</option>
  </list>
  ```

  In this structure, ?? is the *value* attribute of a given <option> element within the <select>, and ???? is the text to go along with the option.

- TextboxArea – This populates a textbox or textarea with the response from the server.

- WindowOpener – Opens a new window and populates it with the response from the server.

- XSLT – Allows for doing XSL transformations on the response from the server, which is assumed to be XML of course. Note that this requires the Sarissa library, not included with APT or JWP.

- FormManipulator - This handler populates one or more fields in a form. Also, it is capable of manipulating the properties of the form and the elements within it. The server process, which you are responsible for writing, must return XML in the following form:

  ```
  <form name="AA">
    <formproperty name="BB" value="CC"/>
    <element name="DD" value="EE">
      <property name="FF" value="GG"/>
    </element>
  </form>
  ```

  ...where AA must be the name of the form, BB is the name of the form property (e.g., action) and CC it's new value. DD must be the name of the element and EE the value you wish to assign to it. FF is like BB but then for the element's property (e.g., disabled or style.width) and GG the new value for the property. As you can see it is possible to connect properties with a dot as long as it is possible in JavaScript to retrieve them with element[prop1][prop2]! This handler is useful for doing form validation. It allows you to return a list of form fields to highlight, values to insert into it, and so forth.

And last, let's see the error handlers that come standard with APT:

- AlertErrorHandler - Pops a JavaScript alert() displaying a message formed by taking the response text and status code from the XMLHttpRequest object that processed the event.

As I hope you will agree, what comes built-in to APT is quite powerful and should handle most of the common scenarios developers need AJAX for.

Timed and Manual Events

Something that comes up rather often in AJAX development is the need to fire continuous events to the server, perhaps to poll the status of a submitted job for instance. APT, being an event-driven paradigm, may not seem like the ideal solution for these issues, but it can be!

For timed events, APT offers the <ajax:timer> tag. This tag sets up a timer that will continuously fire an AJAX event, an event that is configured in exactly the same way, using all the same handlers and options, as any other AJAX event in APT. This tag has three attributes:

- ajaxRef (required)- This has the same meaning as the <ajax:event> tag.

- frequency (required)- This is the amount of time, in milliseconds, between event firings. Note that when the timer is started, it will always wait this amount of time before the first firing takes place, even if startOnLoad is "true"

- startOnLoad (optional)- This determines whether a timer will be started automatically when the page loads (when set to "true"), or whether you will take full responsibility for starting the timer (when set to "false"). The later is good if you want to start the timed event after something else on the page happens, like only start updating a status display after the user clicks a "Start" button for instance. In this case, you will need to call the function that starts the timer yourself.

This tag also will render two JavaScript functions named startXXXX() and stopXXXX(), where XXXX is the *ajaxRef*, with the slash replaced with an underscore. For example, for *ajaxRef* "MyPage/MyButton", the functions rendered would be startMyPage_MyButton() and stopMyPage_MyButton(). As their names imply, these allow you to manually start and stop the timed event. Note that if the stop function is called while an AJAX request is in progress, that event will complete, but no further events will fire unless you call the start function.

Manual events are for those times when you don't want to use the event-oriented paradigm. If you just want a function you can call to fire an AJAX request, yet you still want to use the declarative nature of APT, the <ajax:manual> tag is the answer. It has the following attributes:

- ajaxRef (required)- This is the *ajaxRef* of the <element> in the ajax-config.xml file to apply to this manual function, in the form xxxx/yyyy.

- function (required)- This is the name of the JavaScript function that will be created for you to call. You can then call this function at any time to fire the configured AJAX event.

Writing Custom Handlers

As powerful as APT is out-of-the-box, we fully recognize that sometimes you will need more. Although APT is meant to be a declarative, no-coding AJAX approach, writing custom handlers is the one place you will in fact have to write some code. Even still, because you use these custom handlers in the same way you do the standard handlers in terms of configuration, their usage is still a declarative approach at least.

Writing a custom handler is really quite simple. It requires only that you write a JavaScript function with a particular signature. That signature is different depending on which of the three types of handlers you are writing (request, response or error).

Request Handlers

Here is the signature for a request handler:

```
function myCustomRequestHandler(evtDef) { }
```

The *evtDef* parameter is an associative array that contains the following member:

- resHandParams - This is an array that itself contains three elements in this order: the *type* (name) of the response handler, the *parameter* for the response handler, and the *matchPattern* for the response handler. In this way, multiple response handlers can be used for a given event.

- theForm - The name (or DOM ID) of the HTML form the request handler will use, or null if none.

- targURI - The URI the request will be submitted to.

- reqHandler - The *type* (name) of the request handler this event will use.

- reqParam - The *parameter* for the request handler.

- httpMeth - The HTTP method that this handler will use.

- ajaxRef - The *ajaxRef* of the element firing this event.

- async – "true" if this request will be fired asynchronously, "false" if not.

- evtType - The type of the event, i.e., "onclick", "onblur", etc.

- preProc - The name of the JavaScript pre-processing function to be called before the request is made to the server.

- postProc - The name of the JavaScript post-processing function to be called after the request is received from the server, and after all response handlers have fired, and after all <script> blocks in the response have been executed.

As a simple example, let's look at the code for the standard SimpleRequest handler:

```
function StdSimpleRequest(evtDef) {
  ajaxPartsTaglib.ajaxRequestSender(evtDef, null, null, null, null, null);
}
```

In this case, we see a new facility being used that is available to your own request handlers: the ajaxRequestSender() function of the ajaxPartsTaglib object. Calling this function allows you to not have to worry about the details of making the AJAX request itself, or dealing with the response, or concurrency, or anything else. You simply, in your own handler function, worry about creating the request to go to the server, which means constructing a query string, or POST body, or both.

The ajaxRequestSender function requires a couple of bits of data to do its work, and here is its signature:

```
ajaxPartsTaglib.ajaxRequestSender(evtDef, pb, qs, xhr, headers)
```

The meanings of the parameters are as follows:

- evtDef - The event definition object for the event, passed in to the request handler function.

- pb - XML DOM (or any other content) to submit via POST body, null if none.

- qs - Query string to append, or null if none.

- xhr - This is an instance of an XMLHttpRequest object, or null. Usually, callers of ajaxPartsTaglib.ajaxRequestSender() will pass null for this parameter, in which case a new XMLHttpRequest object will be instantiated to service the call. However, in some cases, you may want to manipulate the object yourself prior to its use. This allows you the opportunity to do that. You may also call ajaxPartsTaglib.getXHR() to get a new instance of XMLHttpRequest, which you can then work with as you see fit and pass along to ajaxPartsTaglib.ajaxRequestSender(). This is recommended as it deals with cross-browser instantiation issues.

- headers - This is an associative array keyed by header name. Because you cannot call setRequestHeaders() on the XMLHttpRequest object until open() has been called, and since this will happen as the last thing ajaxPartsTaglib.ajaxRequestSender() does, it would be impossible to set request headers before then. Some handlers, such as the std:Poster, need to do this however. This argument allows you to do so. Just after open() is called on the XMLHttpRequest instance, this array will be iterated over and the headers it specified, if any, set on the object.

It is highly recommended that you take a look at the source for APT, specifically the .js resource files that contain the standard request, response and error handlers to see all of this in use. These handlers are registered internally in APT, not using the <handler> definition mechanism in the config file, but aside from that, they work exactly as your own custom handlers would.

Response Handlers
A response handler is very similar, but the signature is different. That signature is:

function myCustomResponseHandler(ajCall, resParam)

The *ajCall* parameter is again an associative array which contains the following elements:

- xhr - This is the XMLHttpRequest object instance that handled this request.

- evtDef - This is the *evtDef*, as described above in discussing writing request handlers section, that contains all the configuration information for this event.

- pb - This is the POST body that was POST'd to the target URL, if any.

- qs - This is the query string that was sent to the target URL, if any.

As a simple example, let's look at the code for the standard Alerter handler:

```
function StdAlerter(ajCall, resParam) {
  alert(ajCall.xhr.responseText);
}
```

Again, pretty simple stuff.

Error Handlers
Error handlers are very simple things, with the following signature:

```
function myCustomErrorHandler(ajCall)
```

The *ajCall* parameter is the same as described above in discussing writing response handlers.

In the case of all request and response handlers (but not error handlers at present- expect this to change in a future release to eliminate the need for registration at all), one final step is required to make it all work… the function has to essentially be "registered" with APT in the client. To do this, the following line should appear immediately after your handler function:

```
ajaxPartsTaglib.regXXXHandler("yyy", zzz);
```

Replace XXX with either "req" for a request handler or "res" for a response handler. yyy is the name of the handler, matching the *name* attribute of the <handler> element in the config file, and zzz is a reference to the function itself.

To put is all together, the aptexample2 webapp demonstrates three custom handlers, one of each type. The request handler looks at the value of a <select> field, and based on it, alters the target URL of the AJAX request (this is **not** something we would typical recommend doing, but for demonstration purposes its fine). The response handler inserts the response into a , and also changes the color of the text based on what value was selected. Lastly, the third element in the select will generate a 404 error, and a custom error handler that popups an elert() is demonstrated. Note that the request and response handlers are imported into the page, while the error handler uses the "local" configuration value.

Further Examples

The examples here are obviously very simplistic. Seeing APT in action in a more complex role is where the fun begins! A number of examples exist that will be of interest…

- Practical Ajax Projects With Java Technologies, a book published by Apress in 2006 (Frank W. Zammetti, ISBN 1-59059-695-1). This book is about AJAX in general, shown in real-world example applications. Two of the applications use APT and serve to demonstrate it a bit more fully than these simple examples (please do note however that the applications do not use the latest version of APT… while there are differences, the basic concepts are substantially the same, so you should be able to get nearly as much out of the examples as if they had used the latest version).

- The JWP sample application. When you download JWP, you get a full webapp demonstrating all facets of JWP, including APT. The page in the sample app demonstrating APT shows quite a few different usage scenarios, and stretches the boundaries a little bit. This obviously does use the latest version, so it is well worth checking out. You may find a lot to like about JWP in general too!

- The JWP cookbook. This is a collection of sample webapps for JWP demonstrating some common usages. One of the examples currently available is using APT to do a double-select, i.e., when the user changes one <select>, the contents of another are altered via AJAX.

- Rick Reumann wrote a very nice introductory article here: http://www.learntechnology.net/struts-ajax-crud.do Note however that at the time this article was written, APT was called AjaxTags, so don't let that confuse you! Again, while this does not show the latest version of APT, it definitely does a good job of getting the concepts across.

Conclusion

As I think you can see, APT offers a great deal of power for Java web developers with very little difficulty. Built on a declarative foundation and utilizing an event-driven paradigm, APT allows you to implement AJAX into your applications, whether existing or new, with a minimum of fuss and effort. By the way, APT works just great with most of the popular framework today, including Struts and Webwork! Although APT generally

says almost nothing about what happens on the server-side of the AJAX equation, it frees you from having to worry about the client-side of it, allowing you to concentrate your efforts where, as a Java developer, your expertise can be best applied, and where your existing skillset is focused. But, when the time comes that you need more flexibility and want to tackle some trickier client-side work, APT is flexible enough to not stand in your way.

There are plenty of choices in AJAX libraries these days, and I hope you will give due consideration to APT alongside all the rest. It may not be the ideal choice in all cases, but I believe you will find it is a good choice for many, if not most.

Thanks for reading!

About the author

Frank W. Zammetti is a Web Architect Specialist for a leading worldwide financial institution during the day and a PocketPC/open-source developer at night. He is the Founder and Chief Software Architect of Omnytex Technologies. He has over 10 years of professional development experience, and nearly 15 more of "amateur" experience, and he has been developing web-based applications (Intranet applications mostly) almost exclusively for nearly 8 years. Frank holds numerous certifications including SCJP, MCSD, CNA, i-Net+, A+, CIW, MCP and numerous BrainBench certifications. He is a contributor to a number of open-source projects, including DataVision (http://datavision.sourceforge.net) and PocketFrog (http://pocketfrog.sourceforge.net), as well as having started two: Java Web Parts (http://javawebparts.sourceforge.net) and the Struts Web Services Enablement Project (http://sourceforge.net/projects/strutsws). Frank's resume is available online at http://www.zammetti.com/resume.

Sample webapps: http://www.omnytex.com/articles/apt/aptexample1.zip and http://www.omnytex.com/articles/apt/aptexample2.zip

PDF copy of this article: http://www.omnytex.com/articles/apt/apt.pdf

Microsoft Word copy of this article: http://www.omnytex.com/articles/apt/apt.doc