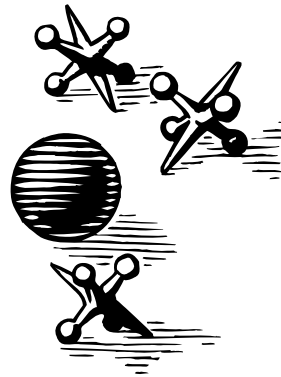


# Tagger Cat's Ajax Processing

January 2006

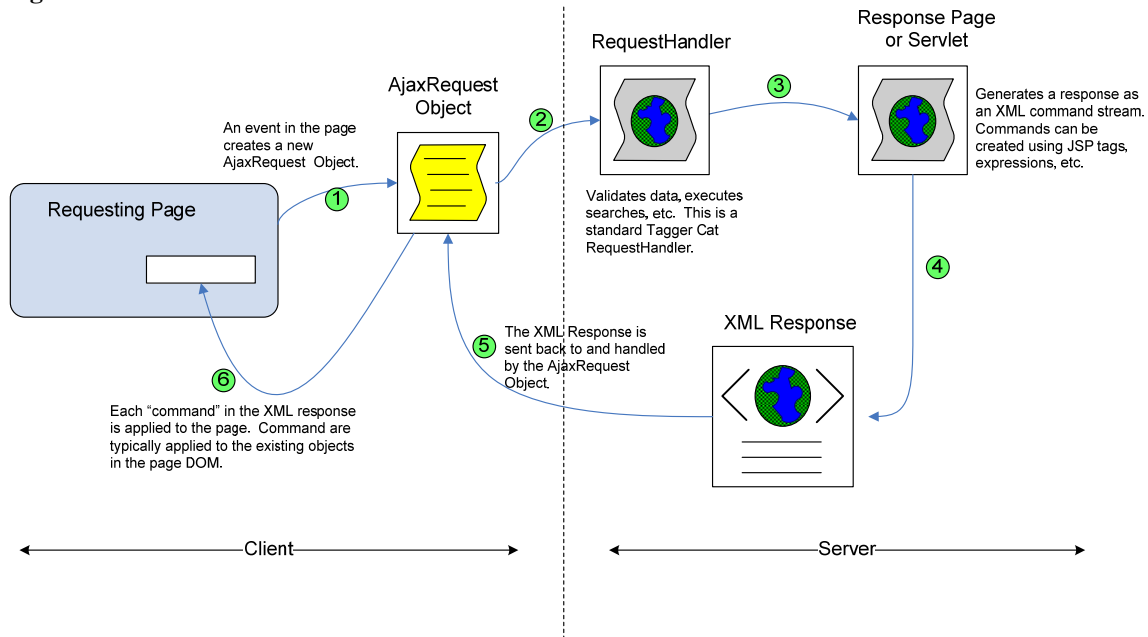


# Ajax Processing with Tagger Cat

Ajax request processing is very similar to standard HTTP request processing. The primary difference is that the Ajax response typically updates only a portion of the page, rather than a complete page refresh on the client. With regular web request processing you can think in terms of the *browser* making a request for an entire new page with each action. With Ajax, you can think in terms of the *page* making a request for a new snippet of page content. This difference is evident when you try to use the browser's back button. With regular web requests your browser records the page history, and you can generally use the back button. Whereas, with Ajax your browser's back button functionality is not aware of the underlying calls being made by the page.

In TaggerCat; Ajax request processing is encapsulated into a single JavaScript class appropriately called: AjaxRequest.

Figure 1



## The Request Processing Steps:

Tagger Cat's Ajax request processing is shown in the diagram above with each main step indicated with a green colored number node.

The response is an XML stream that contains a list of command elements. The response commands are automatically applied to the user's page. The commands are very general, and will cover a wide variety of use cases.

In most cases, JSP page developers do not need to be experts on Ajax, DHTML, JavaScript, etc. to make use of this functionality. For a large number of use cases developers will simply be able to select the content to be applied for an Ajax response directly by using Tagger Cat's Dreamweaver Extension ( The Tagger ).

The processing steps of the Ajax call and response handling are described below. The step numbers correspond to the numbers in the above diagram.

### **Step # 1. A page event initiates the Ajax call.**

The process is initiated from an event on the page; possibly by a field losing focus, a value changing, a selection changes, etc.

A new instance of an AjaxRequest object is created to initiate the Ajax call. The same object instance handles the processing of the response.

The AjaxRequest object is a custom JavaScript class. The AjaxRequest object prepares and sends the Ajax request to the server. Additional request parameters can be added to the AjaxRequest object using the **addParameter**( name, value ) method.. Any added parameters are then included in the subsequent Ajax HTTP request.

The HTTP request is typically sent to Tagger Cat's controller servlet and handled by a Tagger Cat request handler. The actual Tagger Cat request handler used to handle the request could be one of the standard request handlers, or it could be a custom request handler. However, by the nature of Ajax calls, the request handler will more typically be a custom one.

Additionally, just like with regular request processing, you can use Request Handler chaining. Of course, you can also use Method Call request handlers; you don't need to implement full scale request handlers for Ajax requests; the choice is yours.

Therefore, there is nothing at all special required for performing the actions specified in Ajax calls. Ajax calls are no different from that of any other server request.

Also, just like in standard Tagger Cat processing, you can use a separate page for "success" and "fail" results from the request handler.

### **Step # 2. The Ajax Request object formats and sends an HttpRequest.**

Tagger Cat's standard controller functionality is to forward the request to a response page or other resource. The other resource could be a servlet, etc. However, we recommend you use JSP pages for the response resource.

### **Step # 3. The HttpRequest is handled by a Tagger Cat Request Handler.**

The response to the Ajax call is typically an XML document. However, like handling most other requests in Tagger Cat, the generation of the response is most easily handled using a JSP page. Using a JSP page to generate the XML response enables you to use all the standard Tagger Cat functionality. You can use data sources, JSP Tags, expressions, JSTL etc. Depending on the nature of the Ajax call, you can usually use the Dreamweaver Tagger, and templates to help generate the needed response components.

### **Step # 4. The XML response stream is generated.**

The JSP page generates the XML response document specifying the response type to be :

```
<%@ page contentType="application/xml" %><?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

There are two important points to note in the above example:

1. There is no break ( CR-LF) before the <?xml .../> element.
2. The encoding is set to UTF-8.

The JSP file then includes the XML response tags as template content.

The parameters to the AjaxRequest object will typically include the DOM ID of the object(s) on the target page that the response commands will be applied to.

### **Step # 5. The XML response stream is returned.**

The XML response is sent back to the calling AjaxRequest object.

### **Step # 6. The Request Object processes the response as a series of commands.**

The AjaxRequest object ( the same object instance that originated the request ) processes the XML response as a series of commands. The commands are typically applied directly to the DOM of the source page without any custom code required. The target elements on the page are identified by their id attribute. By its nature, the executeScript command is not automatically applied to a DOM element.

### ***Available XML response commands:***

***<setAttribute >***

This command sets the named attribute of the target DOM element. Any number of **setAttribute** elements can be included in the response.

### **<setInnerHTML>**

This command sets the body content of the target DOM element to that specified by the command. This is probably the simplest and most efficient way to change significant page content. Any number of **setInnerHTML** elements can be included in the response stream.

### **<setOuterHTML >**

This command replaces the entire target DOM element with the content specified by the command. Any number of **setOuterHTML** elements can be included in the response stream. Be aware that, as with the **innerHTML**, the **outerHTML** property is not editable on all DOM elements and browser implementations vary.

### **<tbody>**

This creates a new table body element. The `<tbody >` element usually has child `<tr>` and `<td>` elements.

### **<script >**

This command runs the specified script on the client. Any valid JavaScript can be specified. Any number of **script** elements can be included in the response stream.

## **Escaping DOM markup:**

You must escape markup that will be applied to the DOM. Use the `<tc:escape />` JSP tag as shown in the examples.

## **Debugging Ajax Calls**

Debugging your Ajax calls can be both frustrating and challenging. The most significant issues you are likely to encounter are related to cross-browser inconsistencies in DOM APIs.

The good news is that debugging the processing of the Ajax calls on the server is no different from debugging regular Tagger Cat Request handler processing.

The most effective tool for debugging Ajax on the client side is the Firebug FireFox extension. You can find out more about Firebug from: <http://www.getfirebug.com/>.

The `AjaxRequest` class also has some built in functions to help debug Ajax calls on the client side.

Setting `debug` on in the `AjaxRequest` object will show you the request details that are being sent to the server. Then on the response, each command is displayed in an **alert()** window before it is executed ( applied to the DOM ).

On the client side, you should have a DOM inspector installed in your browser. Using the DOM inspector you can view the DOM pre and post Ajax calls.

If you are debugging an issue specific to Internet Explorer, you need to make sure debugging is enabled. To do so:

- 1) Start IE
- 2) Choose Tools → Internet Options to display the Internet Options Dialog box.
- 3) Click the Advanced tab.
- 4) If present, clear the checkbox on the Disable Script Debugging option.

To set a break point in your JavaScript code, or even in the AjaxRequest object itself, insert a break statement: **debugger;** When the **debugger;** statement is executed, the debugger of your choice will be launched.

## Using the AjaxRequest Object

A sample usage of the AjaxRequest object is shown below.

```
1. function suggestName() {
2.     var ajr = new AjaxRequest();
3.     ajr.setDebug( document.getElementById( "debugAjaxId" ).checked );
4.     ajr.setUpdateElementId( 'suggestions' );
5.     ajr.setActionName( 'sessionScope.MyEventHandler.doit' );
6.     ajr.setResponsePage( '/AjaxExamples/AjaxSuggestResponse.jsp' );
7.     ajr.addParameter( 'customerName', document.getElementById(
   'customerName' ).value );
8.     ajr.send();
9. }
```

This function is called from an event handler on the page. Notable code in this example is:

**Line 4:** Sets the id of the DOM element to be updated with the request. This element id is then included as a parameter in the subsequent Http request with the name **updateElementId**. The DOM element id is typically used in the response to identify the element ID that commands apply to.

**Line 5:** Sets the **requestName** request parameter. Tagger Cat's controller then uses that name to determine the request handler(s) that will process the request; just as it does for any other request.

**Line 6:** Sets the name of the **responsePage** request parameter. This is the name of the JSP page that will generate the response commands.

**Line 7:** Sets an additional request parameter. In this example, the **customerName** field is the form element used to enter the customer name. The field's current value is used in

the Ajax call to filter the suggestion list to include only those values with the same leading characters.

## ***Ajax Examples:***

Refer to the Western Auto sample application for more details of these examples and complete source code listings.

### **Example 1: Validating an expression**

This example demonstrates how to make an Ajax call to validate a field entry. Specifically, in this example, the field entry is an expression.

#### **The Client Side:**

On the client side the components are:

1) The form field that holds the expression that is being validated. In this example, the expression will be validated when the field loses focus.

```
<label>Expression
    <input type="text" name="textfield" tabindex="1" size="60"
onBlur="validateExpression('expFieldId')">
</label>
```

2) The HTML element that is used to display a validation message:

```
<span id="reportExpressionErrorId" ></span>
```

3) The JavaScript method used to validate the expression; in response to the **onBlur()** event handler on the expression field.

```
1. <script type="text/javascript" >
2.
3. function validateExpression( valueElementId ) {
4.     var ajr = new AjaxRequest();
5.     ajr.setDebug( document.getElementById( "debugAjaxId" ).checked );
6.     ajr.setRequestName( 'sessionScope.MyEventHandler.validateExpression'
7. );
8.     ajr.setResponsePage( '/AjaxExamples/AjaxExpEvalResponse.jsp' );
9.     ajr.addParameter( 'expression', document.getElementById(
10. valueElementId ).value );
11.     ajr.send();
12. }
```

#### **The Server Side:**

On the server side, the components are:

1) A request handler that validates the expression.

The request handler parses the expression, and sets a request attribute named `isExpressionOk` indicating the validity of the expression's syntax. If there is an expression error, then an error message is set in a request attribute named `syntaxErrorMessage`. See the sample application for the actual code listing.

2) A response JSP page generates the response commands as an XML stream. Note, that Tagger Cat's tag library is used to generate the conditional content. The source code for `/AjaxExamples/AjaxExpEvalResponse.jsp` is shown below.

```
1. <%@ page contentType="application/xml" %><?xml version="1.0"
   encoding="UTF-8" standalone="yes" ?>
2. <%@ taglib prefix="tc" uri="http://www.taggercat.com/taglibs/taggercat"
   %>
3. <ajaxResponse>
4.   <elementToUpdate targetId="reportExpressionErrorId" >
5.     <setInnerHTML>
6.       <tc:escape>
7.         <tc:ifThenElse test="request.isExpressionOk" >
8.           <tc:then>
9.             <p><b>Expression syntax is valid</b></p>
10.          </tc:then >
11.          <tc:else>
12.            <p>
13.              <tc:out value="request.syntaxErrorMessage" /> </p>
14.            </tc:else >
15.          </tc:ifThenElse >
16.        </tc:escape>
17.      </setInnerHTML>
18.    </elementToUpdate>
19. </ajaxResponse>
```

## The Results:

When the expression is valid, the message element changes to:

```
<span id="reportExpressionErrorId" ><p><b>Expression syntax is
valid</b></p></span>
```

When the expression is not valid, the message element changes to include an image tag indicating the error condition and the error message.

```
<span id="reportExpressionErrorId" ><p>
  An operator was expected after 200</p>
</span>
```

## Example 2: Suggestion List

This example demonstrates how to make an Ajax call to provide a suggestion list for a field. In this example, a search field is provided with a list of customer names.

### The Client Side:

On the client side the components are:

1) The form field that the suggestion list will be provided for. In this example, the suggestion list will be provided with each change to the input field. For example, if you type in the letters "Sa" into the field, customers names with that matching prefix will be shown in the suggestion list.

```
<label>Customer Name:
    <input id="customerName" type="text" name="textfield" tabindex="1"
size="30" onKeyUp="suggestName();" >
</label>
```

2) The DIV element where the suggestion list will be shown within.

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

3) The JavaScript functions to make the Ajax call, and another one to handle a selection event within the suggestion list.

```
1. <script type="text/javascript" >
2.
3. function suggestName() {
4.     var ajr = new AjaxRequest();
5.     ajr.setDebug( document.getElementById( "debugAjaxId" ).checked );
6.     ajr.setUpdateElementId( 'suggestions' );
7.     ajr.setActionName( 'sessionScope.MyEventHandler.doit' );
8.     ajr.setResponsePage( '/AjaxExamples/AjaxSuggestResponse.jsp' );
9.     ajr.addParameter( 'customerName', document.getElementById(
    'customerName' ).value );
10.    ajr.send();
11. }
12.
13. function nameSelected( list ) {
14.    document.getElementById( 'customerName' ).value = list.options[
    list.selectedIndex ].text;
15.    document.getElementById( 'suggestions' ).innerHTML = "";
16. }
17.
18. </script>
```

The **nameSelected()** function is called when there is a selection in the suggestion list. It does two things:

1) it copies the selected item into the Customer Name form field

- 2) it removes the generated suggestion list from its parent DIV element

## The Server Side:

On the server side, the components are:

- 1) A request handler that validates the expression. In this example, the `doit()` method on the `sessionScope.MyEventHandler` is called. There is really no functionality in this method other than setting no cache header values on the HTTP response object.
- 2) A response JSP page generates the response commands as an XML stream. Note, that Tagger Cat's tag library is used to generate the content. The source code for `/AjaxExamples/AjaxSuggestResponse.jsp` is shown below

```
1. <%@ page contentType="application/xml" %><?xml version="1.0"
   encoding="UTF-8" standalone="yes" ?>
2. <%@ taglib prefix="tc" uri="http://www.taggercat.com/taglibs/taggercat"
   %>
3. <tc:datasource name="scustomers" busObject="CUSTOMERS"
   instanceVar="scustomers" mode="deferred" whereClause="" orderBy="Name"
   refresh="false" />
4. <ajaxResponse>
5.     <elementToUpdate targetId="suggestions" >
6.         <setInnerHTML>
7.             <tc:escape>
8.                 <select name="select" size="4" style="width:120px"
   onChange="nameSelected(this);">
9.                 <tc:dsOptionList dsn='scustomers' displayField='Name'
   codeField='CustNum'
10.                condition="upper(scustomers.Name) LIKE upper(param.customerName)+'*'"
   />
11.            </select>
12.        </tc:escape>
13.    </setInnerHTML>
14. </elementToUpdate>
15.</ajaxResponse>
```

Points to note in this code are:

Line 3: A separate data source name “scustomers” is declared.

Lines 8 – 11: A HTML select element is created using the `tc:dsOptionList` tag.

Line 10: The condition attribute is set to filter only records that match `param.customerName` value. Where the `param.customerName` value is the text the user has entered into the Customer Name form field. This condition is an expression evaluator expression, and not a SQL where clause expression. Using the expression evaluator allows us to perform the SQL query once, and then filter the rows internally.

## Using the tcAjaxAction JavaScript function

The client side examples shown above have made use of a custom JavaScript function specific to the task. Although these functions are quite straight forward, you may want to have something a bit more generic.

The **tcAjaxAction** function can be used as a single statement to make an Ajax call. Here is an example:

```
<tc:actionButton value="Where Used?" action="antBuild" dsn="Template"
    onclick=
"tcAjaxAction('antBuild','/Ajax/AjaxAntCallResponse.jsp',
{targetElement:'whereIsTemplateUsedId',
buildFile:'WhereUsedBuild.xml',templateName:'${Template.name}'} )"
title="Where is this template used?." />
```

Take note of the last parameter in the function; it is called an object literal, or an associative array. It is written as { property1 : value1, property2 : value2, ... }.

When used with the **tcAjaxAction** function, each property becomes a request parameter in the Ajax call. Also, note that the value of the second property in the object literal is set using a JSP expression: `'${Template.name}'`.

In this example, the **tcAjaxAction** function invokes the request handler mapped to the “antBuild” action name. Of course, you could just as easily call any server side bean method; for example:

```
tcAjaxAction('MyActionBean.createNewAccount','/Ajax/AjaxAntCallResponse.jsp',...)
```

Therefore, calling server side Java methods from JavaScript is quite trivial.