Ajax Proposals

Table of content

- Background
- Client Side Support
- Proposals
  1. Page actions
  2. Ajax Controls
  3. Ajax Behaviors
- Which one?

Background

I'd like to see improved Ajax support in Click 2.2.0, especially around the area of reusable Ajax controls. It is not currently possible to create such controls.

There are a number of approaches to improve Ajax support, so I've come up with a list of proposals to stimulate further discussion. A final solution could combine some of the proposals.

I'll use jQuery where client-side JavaScript is required to convey a concept.

Client Side Support

There are two sides to Ajax support, the client side HTML markup and JavaScript code to initiate the Ajax request, and the server side support, to handle the request and return a partial response.

In this document I won't go into great detail of the client-side support and will instead focus on the server side.

But lets quickly look at the JavaScript code necessary to initiate an Ajax request. Below is a simple client-side example to initiate an Ajax request when a link is clicked.

Please note: the code below is an example of the markup and JavaScript code that must be rendered by a client-side Click implementation. The implementation will most likely use generic templates and that accepts variables such as the CSS selector, event, page url etc.

```
my-page.htm

<a id="mylink" href="my-page.htm">Click Me</a>
<div id="target">Update Me</div>
```

Javascript:
// Bind a click handler to the link with id 'mylink'
$('#mylink').bind('click', function(event) {
    // Make an Ajax call to my-page.htm passing the parameters 'mylink', to identify which
    // control was clicked and 'event', to identify which event was fired.
    // If the request succeeds, the 'success' method is called
    $.get("my-page.htm", { mylink: 1, event: event.type }, success);
    function success(data) {
        // Update the target div with the Ajax response.
        $('#target').html(data);
    }
});

This example is fairly simple and self-explanatory, but lets quickly highlight the main parts.

jQuery provides a **bind** function which allows us to add an event handler to an HTML element. The **bind** function consists of three parts. First you specify a CSS selector which selects the HTML elements to perform the bind operation on. In this example the CSS selector is the ID of the link we want to bind to -> '#mylink' (The hash (#) prefix denotes a CSS ID selector). Then you specify which **event** to bind to. In the example above this is the 'click' event. Lastly you specify the function to invoke when the event is triggered. (jQuery's **bind** is synonymous to JavaScript's addEventListener (attachEvent in IE). Another way to bind is to specify the event handler as an HTML element attribute (<a onclick="success" href="...">Click me!</a>))

The jQuery bind function allows one to bind multiple HTML elements and events to the same function:

```javascript
$('#mylink').bind('click', clickFunction(event)
$('#mybutton').bind('click', clickFunction(event)
```

or one HTML element to multiple events:

```javascript
$('#mylink').bind('click', clickFunction(event)
$('#mylink').bind('hover', hoverFunction(event)
```

From Click's perspective, it needs to render this bind statement along with the JavaScript function that will be invoked when the event is performed.

For client side support, I suggest we keep JavaScript / Ajax functions in Velocity templates instead of wrapping/hiding it in a Java API. One of the advantages of using Velocity is one can immediately see changes to the template, no compilation or redeploy step is required. Also allows Click's Ajax support to be fairly transparent and enables users to easily customize the templates for their own needs. In addition users can use their favorite JavaScript frameworks and plugins without having to learn the equivalent Java API.

Some frameworks prefer to hide JavaScript from developers as they feel the complexity of JavaScript will be overwhelming. Instead a Java API is exposed against which the developers can code. This is great, until it breaks, in which case the developer (remember he/she doesn't know any JavaScript) needs to find a fix. To find a fix the developer will rely heavily on the framework developers for support and in many cases will need to start learning and understanding JavaScript anyway 😞

I acknowledge that JavaScript can be a pain, however I believe most of the problems come from different browser implementations of the specification. With the advent of modern JavaScript libraries such as jQuery, Prototype, YUI, Mootools etc, I believe most of the pain points have been addressed.

Another problem I see with hiding JavaScript through a Java API is that the framework will be in a constant battle to keep up with the feature set of the JavaScript libraries. Consider properly documenting the work and you can see that keeping the Java API's up to date is a non trivial task.

So in summary, I suggest we allow users to write client-side code in JavaScript using their favorite library. Click should be concerned with handling the incoming Ajax request and passing back a partial result object.

Next I'll cover the proposals to handle the Ajax request and response on the server-side.

## Proposals
1. Ajax aware Page actions

The simplest way to improve Ajax support is to allow Page actions to be called directly from the browser. The Page action could return a Partial response that contains the data to return to the browser.

```java
public MyPage extends Page {
    public Partial loadLabel() {
        // A partial object provides a way to return a partial response to the browser.
        // Any content added to the Partial object,
        // is streamed back to the client
        Partial partial = new Partial();
        partial.add(new Label("hello"));
        return partial;
    }
}
```

The example client-side Ajax call will need slight modification, since a Page action is invoked instead of a Control:

```javascript
$.get("my-page.htm", { mylink: 1, event: event.type, pageAction: loadLabel }, success);
```

In order to support invocation of Page actions, Click should be enhanced so that if the 'pageAction' request parameter is present, that request parameter value will be the Page action to invoke. The signature of the method must be: "public Partial methodName()". Except for creating the Page object and onSecurityCheck event, no other page events (onInit, onRender etc) are executed.

Please note that this feature could be useful for non Ajax requests as well. Consider a dynamic <img> element that displays a different image depending on its parameters. The ability to invoke a Page action directly would be useful in this case as the Page action implementation could inspect the request parameters, grab the image from the database and stream back the image to the browser. All this without having to invoke other Page events (onInit, onProcess) or having to set the Page path to null, which often trips new users.

**Pros:**

- simple to implement
- easy for users to understand and use

**Cons:**

- caters only for GET requests, not POSTS, since page events are not executed
- does not cater for ajax aware controls
- interaction with other components on the Page will be fairly limited, since no initialization or binding occurs

2. Ajax aware Controls

A more sophisticated approach would be to allow control listeners to be invoked for Ajax requests. A special AjaxListener can be set as a Control listener, that will be invoked if the incoming request targets that control:
public MyPage extends Page {
    public void onInit() {
        Form form = new Form("form");
        TextField field = new TextField("field");
        field.setActionListener(new AjaxListener() {
            public Partial onAjaxAction(Control resource, Event event) {
                Partial partial = new Partial();
                partial.add(new Label("hello"));
                return partial;
            }
        });
        form.add(field);  // <- First we attach the field to its parent form to ensure the
        field.getId()     // will return its fully qualified ID: "form-field", not just
        "field".
        AjaxHelper helper = new AjaxHelper();
        helper.ajaxify(field); // <- this call will inject necessary Javascript code to make Ajax calls
    }
}

This approach will allow creating reusable ajax aware controls, but it could lead to an explosion of new controls, as each control needing ajax behavior will need to be extended and enhanced. This problem could be alleviated by creating Helper objects that can "inject" the necessary JavaScript code to make Ajax calls from the browser.

The Helper's ajaxify() call could leverage the control's getHeadElements() method in order to add JavaScript libraries, JavaScript template code etc.

Pros:
- Relatively simple to understand and use
Both POST and GET requests are supported. For example an Form could be created which performs an Ajax post. As the onInit event is executed, the Form and Fields will be created and ready to be bound to incoming request parameters. Thus a Submit button can register an Ajax listener and save the submitted Form data to the database, before responding with a Partial response, such as a success message, or error message.

Can create prepackaged Ajax aware controls

If JavaScript is not enabled the Control ActionListener's onAction(Control source) method will be invoked, providing a simple fallback mechanism.

Cons:

- It could encourage writing complex Pages that are hard to understand and maintain
- Could lead to an explosion of control subclasses to add Ajax functionality. This can be alleviated using Helper objects to encapsulate Ajax behavior
- Can lead to difficulties if an Ajax control is added inside another control's listener and subsequent requests won't have the Ajax control present in the server-side component tree
- The stateless nature of Click makes Ajax applications inherently tricky, because if a non-Ajax request is made to the server, all Ajax related state will be undone. This can be alleviated by using Stateful pages, but since requests to stateful pages are serialized, this creates another, albeit small, problem in that parallel Ajax requests to a stateful page won't work. In other words if the browser makes two Ajax requests to a stateful Page where the first request is very resource intensive and takes 10 seconds to execute, the second Ajax request will have to wait until the first request finishes. Although this might not be a common use case, it is worth taking into consideration.

Outstanding Issues:

- A control can only have a single listener defined. How will a control handle multiple incoming events? For example a field might handle two Ajax events, "onblur" for validation purposes, and "onkeypress" for auto-completion. The Event object passed into the listener could be inspected, but this leads to if-else logic. This brings up the idea of having multiple listeners per control.
  - **Pro**: Each listener could be registered for a specific event, and will only be triggered for Ajax requests of that event type.
  - **Con**: Having multiple listeners per control could increase the conceptual complexity and could lead to an increase in maintenance. For example, if the client-side code neglects to pass the event parameter, no listener will be invoked. The user will now have to deal with whether the control ID or event request parameter or both are missing.

3. Ajax Behaviors

Behaviors is a concept borrowed from Wicket and JSF. A behavior is attached to a Control and will be notified by the Control for specific events such as oninit, onProcess, onRender, getHeadElements. A behavior can also act as a listener and receive events, just like a Control does.

In terms of Ajax, a Behavior could be added to a Control and will inject the necessary JavaScript and Ajax code needed by the control. Further, the Behavior will act as a listener and will receive the incoming request. So unlike Proposal 2 above, the Control will not be invoked for Ajax requests.

To make things more concrete, let's look at an example:
public class MyPage extends Page {

    public void onInit() {
        Form form = new Form("form");
        TextField field = new TextField("field");

        AjaxBehavior behavior = new AjaxBehavior("click") {
            // Note: Behavior has an onAction event similar to an ActionListener
            public Partial onAction(Control resource, Event event) {
                Partial partial = new Partial();
                partial.add(new Label("hello"));
                return partial;
            }
        };

        field.addBehavior(behavior);
        form.add(field);  // <- First we attach the field to its parent form to ensure the
        field.getId() // will return its fully qualified ID: "form-field", not just
        "field".
    }
}

Pros:

- Won't lead to an explosion of new controls since the Ajax behavior is encapsulated inside a reusable object. That said certain controls, such as Form, might still benefit from an Ajax specific subclass, however this should be avoided as much as possible so that existing Form subclasses can add Ajax behavior.
- If JavaScript is not enabled the Control ActionListener's onAction(Control source) method will be invoked, providing a simple fallback mechanism.
- Both POST and GET requests are supported. For example an Form could be created which performs an Ajax post. As the onInit event is executed, the Form and Fields will be created and ready to be bound to incoming request parameters. Thus a Submit button can register an Ajax listener and save the submitted Form data to the database, before responding with a Partial response, such as a success message, or error message.
- Can create prepackaged Ajax behaviors and Ajax aware controls

Cons:

- It could encourage writing complex Pages that are hard to understand and maintain
- The concept of a Behavior is a bit complex to understand at first.

Outstanding Issues:

- The initial thinking is that a behavior can only have a single listener defined. How will a behavior handle multiple incoming events? For example two behaviors could be registered on a field, "onblur" for validation purposes, and "onkeypress" for auto-completion. The Event object passed into the listener could be inspected, but this leads to if-else logic. This brings up the idea of having multiple listeners per behavior.
  - **Pro**: Each listener could be registered for a specific event, and will only be triggered for Ajax requests of that event type.
  - **Con**: Having multiple listeners per behavior could increase the conceptual complexity and could lead to an increase in maintenance. For example, if the client-side code neglects to pass the event parameter, no listener will be invoked. The user will now have to deal with whether the control ID or event request parameter or both are missing.

Which one?

I'm leaning towards introducing both Page actions and Ajax Behaviors. The simplicity of Page actions combined with the flexibility and reusability of Ajax Behaviors seems like a good choice.