41RenderCycleDesign

NOTE: This page refers to an older version of Tapestry. For current information on Tapestry's AJAX support, see http://tapestry.apache.org/ajax-javascript.html.

What "ajax" means to Tapestry

There are many different scenerios and design choices for using ajax within web applications, but I think most of them can still be condensed into three basic types of communication:

- **Asynchronous client requests** with textual/xml return data from server. These are the most common types of requests. Ones that tacos primarily uses for almost all of its operations. At it's core it could be used for a million different things. Tacos uses this paradigm to manipulate what looks like a normal tapestry request into an xml document that is parseable and manageable on the client.
- **Asynchronous JSON requests** - These are requests like any other, but expect a very specific type of data returned from the server, data that is validily structured as http://json.org data. Such a structure describes a javascript object.
- **Comet/Server data push** - The term originally coined by alex from http://dojotoolkit.org, describes the process of pushing data from the server to a client. This enables a slightly more dynamic user interface that doesn't involve having to constantly poll servers for new data, but it has the added drawback of having long lived open connections.

Asynchrnous Requests

These types of requests can be anything that a normal http request/response can be, except that in the case of tapestry you commonly want to invoke or update a certain portion of your page dynamically. An example below will illustrate this usage.

This example is a scenerio where we have a PropertySelection list that allows a user to select a "Project" that they would like to work on, when they make a selection in the list we would like the server to update another component on our page, "ProjectDetails". This component displays detailed meta information about our chosen project. Since the UI is such that we would like to not have to refresh the entire page whenever one of these selections is made we will use "ajax" to refresh only the "ProjectDetails" component when the selection is made. Here is a visual example:

? Unknown Attachment

In the sample given we will have to create a tapestry request that only renders the output from the ProjectDetails component. In it's most basic form that would look something like:

? Unknown Attachment

Tacos ASYNC Requests

In order to get the behaviour of "selecting" a particular option on a PropertySelect component, as well as refreshing a specific component on a page - all with tapestry correctly remembering whatever state was selected on the component in both server and client - we must create a valid request. Using dojo this is not very hard. We can do something like this to the PropertySelect html to cause the initial request to go out:

```
<select name="projectSelect" >
<item value="0">Select</item>
<item value="proj1">Project 1</item>
<item value="proj2">Project 2</item>
<item value="proj3">Project 3</item>
</select>

<script type="text/javascript">
 dojo.event.connect(dojo.byId("projectSelect"), "onchange", function() {
   dojo.io.bind({ url:  "http://localhost:8080/app?
service=direct&component=projectSelect&listener=selectProject",
  load:function(data) {
        //do something with returned data here
   }
 });
</script>
```

This sort of request, at least in tacos, would return a response to the client looking somewhat like:

```
<xml>
  <response-element id="projectSelect">
     <select name="projectSelect" >
<item value="0">Select</item>
<item value="proj1" selected="true">Project 1</item>
<item value="proj2">Project 2</item>
<item value="proj3">Project 3</item>
</select>
  </response-element>
  <response-element id="scriptblock">
      dojo.event.connect(dojo.byId("projectSelect"), "onchange", function() {
   dojo.io.bind({ url:  "http://localhost:8080/app?
service=direct&component=projectSelect&listener=selectProject",
  load:function(data) {
        //do something with returned data here
   }
 });
</script>
  </response-element>
</xml>
```

As you can see, all of the same issues/solutions put in place for html content have to be similarly applied to javascript content. If you allowed the Body component to render the javascript block for a typical response you would get an extroidinary amount of script back in your response that had nothing to do with the components you were trying to refresh. A new interface needs to be added in that will allow similar "filtering" to occurr with javascript blocks, on a per component basis, as with normal html content.

What's happening on the server?

In it's most basic form, tapestry as a whole generally does the following.

- Takes an incoming HttpServletRequest and sets up all of the core tapestry infrastructure, such as hivemind injectable objects for WebRequest/etc.
- Looks at the parameters for the incoming request, ie "&service=direct" to determine which service should handle the request. In this case we want the DirectService, which is more or less the staple of most system requests.
- The direct service reads in the request parameters and invokes the "listener" that was specified in the request, once that listeners method has been invoked it then hands off the response to the ResponseRenderer service.
- The ResponseRenderer sets up the actual IMarkupWriter which is tied to the OutputStream being written back to the client browser (in most cases).
- Once the IMarkupWriter is setup all request flow is handed off to the IRequestCycle object via IRequestCycle.renderPage(IPage page); (I left out the part about pages, but suffice it to say that the DirectService is able to determine which page a particular response should go to and sets it up accordingly)
- The Page object, which probably in 90% of cases is an AbstractPage instance then starts to render itself. This rendering is where ajax requests start to break down.

What we see with Page/Component rendering is a container/child sort of rendering model, where nothing can be known about a particular containers children save by the container itself. So, rendering is handled by components directly. Via a combination of IRender.render() IComponent. renderComponent(writer, request) the chain flows down and down through all of the contained components until the entire response is written.

The problem with this for ajax is that we don't want all the components to render. We only want one, or maybe two or three or whatever. Replacing the IMarkupWriter instance with one of your own doesn't do it because the IMarkupWriter is dumb in that it only knows how to write textual data back, it doesn't know or care whether its currently rendering component A or component Z. It writes whatever markup it's told and that's it.

The tacos solution to this problem was to sort of "hot swap" IMarkupWriters in the rendering chain, depending on whether a component was requested to have it's content renderd by the client. This was achieved by creating two markup writers for every request. One NullMarkupWriter, which when used throws out all of the content that is written to it, and another valid IMarkupWriter that is used to write valid content to the client. The only problem left was how to switch out the correct writer depending on the component being rendered?

The solution was to use javassist to extend "every" component in the system, whether it is tapestrys compnoent or your own custom component via subclassing any object that .isAssignableFrom(AbstractComponent.class). The renderComponent() method was enhanced to look something like this:

```
 // The writer being passed in here is a NullMarkupWriter by default in ajax responses
 public void renderComponent(IMarkupWriter writer, IRequestCycle cycle)
{
 if (!ajaxRequest.isValidRequest())
   super.renderComponent(writer, cycle);

 if (ajaxRequest.componentWasRequestedToRefresh(this))
   super.renderComponent(ajaxRequest.getAjaxWriter(), cycle);
}
```

That's the core of the response model. There are lots of other ways to do it and nuances depending on the problem being solved (ie dealing with javascript, doing pre/during/after response javascript effects/etc..).

The CORE Problem

The largest problem with all of this is the fact that components directly control the IMarkupWriter instance being used to render the response. There are no extension points or other known ways to control which components responses go back to the client and which don't. Adding in new renderComponent() logic that handles JSON responses for instance creates even more problems, because in those instances the IMarkupWriter interface isn't even remotely close to resembling the structure of a JSON response.

The Solution

In order to start getting a better handle on controlling the response, as well as managing the fact that one response might require calling renderComponent (IMarkupWriter, IRequestCycle) and another might require renderComponent(IJSONWriter, IRequestCycle) I have introduced a new object called ResponseBuilder.

This is what the ResponseBuilder interface looks like, which is remarkably similar to ResponseRenderer:

```
public interface ResponseBuilder {

 void renderResponse(IRequestCycle cycle);
}
```

The difference is that the ResponseBuilders that are available are configured from a hivemind configuration point using a Factory to choose the type of builder that is used in a response depending on the header and parameters receieved in a request. For instance, we now currently have a DefaultResponseBuilder and a JSONResponseBuilder. The factory uses a configured list of ResponseContributor entries which are responsible for reading in request meta information and determining which type of response should be given.

The end result might be an object looking like this for default responses:

```
public class DefaultResponseBuilder implements ResponseBuilder {

  protected IMarkupWriter _writer;

  public DefaultResponseBuilder(IMarkupWriter writer) { ... }

  public void renderResponse(IRequestCycle cycle) {
     IPage page = cycle.getPage();
     page.renderResponse(this, cycle);
  }
}
```

That in and of itself isn't very new a concept to put past, as it's more or less already happening that way. The new JSONResponseBuilder does however start to change things.

```
public class JSONResponseBuilder implements ResponseBuilder {

  protected IJSONWriter _writer;

  public JSONResponseBuilder(IJSONWriter writer) { ... }

  public void renderResponse(IRequestCycle cycle) {
     IPage page = cycle.getPage();
     page.prepareForRender();
     page.renderResponse(this, cycle);
  }
}
```

Mirky Waters

The previous solution section outlined what I believe is a good start to the steps needed to enable some of the functionality these ajax/json responses require. Namely being able to either suppress the output in a render cycle or perform entirely different output(as in the case with JSON responses).

An AJAXResponseBuilder implementation would need to work itself into the render() logic in such a way that it could either internally choose to render or not render particular components, or have methods available allowing components to choose or not choose to render themselves. I would rather have the logic be internal to the ResponseBuilder as this is one of the largest gripes I have with tacos currently. It makes it very unclean to do things like this:

```
renderComponent(IMarkupWriter writer, IRequestCycle cycle) {
  AjaxResponse ajax = getAjaxResponse(); //injected

  if (!ajax.isAjaxResponse()) {
    renderJavaScriptOutput();
  }

  if (ajax.isAjaxResponse() && somethingFooIsTrue) {
    ajax.addPreEffectsJavascript("javascript:doPreEffects()");
  }
}
```

The example given is completely made up but more or less shows all of the "checks" that have to go on to correctly maintain server state with client browser state. It would be much nicer to ultimately have something along the lines of:

```
  public class AjaxResponseBuilder implements ResponseBuilder {

  protected IMarkupWriter _writer;
  protedted IMarkupWriter _nullWriter = new NullMarkupWriter();

  protected List updateComponentIds;

  public AjaxResponseBuilder(IMarkupWriter writer) { ... }

  public void renderResponse(IRequestCycle cycle) {
     IPage page = cycle.getPage();
     page.renderResponse(this, cycle);
  }

  public void renderComponent(IComponent component, IRequestCycle cycle) {
      if(updateComponentIds.contains(component.getId()) {
          component.renderComponent(_writer, cycle);
      } else
        component.renderComponent(_nullWriter, cycle);

      if (updateComponentIds.contains(component.getId()) {
          component.renderScriptContributions(_writer, cycle);
      }

      //etc....
  }
}
```

The basic idea is to take out the logic that was previously done via javassist contributions, and at the same time add a layer of additional logic. Such as pre /during/post javascript contributions to the "client request render cycle", as well as a lot of other things I'm sure I'm forgetting.