Tapestry5AndJavaScriptExplained

Introduction

Note: This page refers to Tapestry 5.2 and later. For Tapestry 5.0 and 5.1, just use @IncludeJavascriptLibrary instead of @Import, and RenderSupport instead of JavaScriptSupport

Main Article: JavaScript

JavaScript is a critical element of modern web applications. Historically "real" software developers used to avoid JavaScript either because they had more important things to do, or because they simply didn't want to be bothered with front-end details. Well as we know, times have changed and are changing still. If you want to have an application that can compete with other modern applications, you have to use JavaScript (including Ajax).

Modern frameworks, such as GWT and Ruby on Rails, go a long way in removing the pains of JavaScript from developing web applications. Tapestry 5 is no exception and has come a long way in making JavaScript transparent to developers (not to mention 3rd party libraries).

However, there are those who want to know how it all works. Perhaps you're tracing some weird error, developing a sophisticated component with JavaScript, or maybe you're just curious. This article is for such people, so we'll have a look at how Tapestry 5 elegantly weaves JavaScript into your application by creating a mixin. This mixin can be applied to links (that is, PageLink or ActionLink components) so that the user will be asked to confirm that they would like to follow that link, which is ultimately accomplished by a traditional JavaScript `confirm` dialog.

What You Should Already Know

This article assumes that you have some basic knowledge of Tapestry 5. You don't need intimate knowledge of /how/ it works, just an understanding of how to develop applications with it. In addition you should also be acquainted with the following concepts:

- Component Mixins
- Some JavaScript knowledge. You should at least have a basic understanding of client-side objects and how they are created.
- Understanding of the JSON format and the prototype library would be very helpful, but not required. A good starting point for modern JavaScript programming (prototype oriented, but still generally useful) is http://www.prototypejs.org/learn.

Getting Started

To start our project we'll use the Maven quickstart archetype as described at Getting Started. Assuming you have Maven installed execute the following from a console:

```
mvn archetype:create -DarchetypeGroupId=org.apache.tapestry -DarchetypeArtifactId=quickstart -
DarchetypeVersion=5.0.11 -DgroupId=net.godcode -DartifactId=jsclarity -DpackageName=net.godcode.jsclarity -
Dversion=1.0.0-SNAPSHOT -DarchetypeRepository=http://tapestry.apache.or
```

I use Maven 2, Eclipse, the Maven 2 Eclipse plugin, and Jetty (5) for my development environment. I will make no assumptions about what you use, so when we create a class or run the application I will simply say "create this class" or "run the application." I will however make note of where certain files belong in the context of a Maven project.

Creating the Mixin

Let's create our mixin class. Mixin classes live in the `mixins` sub package of your application.

Maven location: src/main/java/net/godcode/jsclarity/mixins/Confirm.java

```
package net.godcode.jsclarity.mixins;

import org.apache.tapestry5.BindingConstants;
import org.apache.tapestry5.ClientElement;
import org.apache.tapestry5.services.javascript.JavaScriptSupport;
import org.apache.tapestry5.annotations.AfterRender;
import org.apache.tapestry5.annotations.Import;
import org.apache.tapestry5.annotations.InjectContainer;
import org.apache.tapestry5.annotations.Parameter;
import org.apache.tapestry5.ioc.annotations.Inject;

/**
 * A simple mixin for attaching a JavaScript confirmation box to the onclick
 * event of any component that implements ClientElement.
 *
 * @author <a href="mailto:chris@thegodcode.net">Chris Lewis</a> Apr 18, 2008
 */
@Import (library="confirm.js")
public class Confirm {

    @Parameter(value = "Are you sure?", defaultPrefix = BindingConstants.LITERAL)
    private String message;

    @Inject
    private JavaScriptSupport javaScriptSupport;

    @InjectContainer
    private ClientElement element;

    @AfterRender
    public void afterRender() {
            javaScriptSupport.addScript("new Confirm('%s', '%s');", element.getClientId(), message);
    }

}
```

The first thing you may ask is "What does the @Import do"? It includes a JavaScript library, but we'll have a closer look at it later. For now let's look at the a `fterRender` method.

```
        @AfterRender
        public void afterRender() {
                javaScriptSupport.addScript("new Confirm('%s', '%s');", element.getClientId(), this.message);
        }
```

Here we do one thing: insert a little piece of JavaScript in our page. For now don't worry about when/how/where this happens, just understand that this script code will be called after the page has properly loaded in the user's browser. You'll notice that it's calling a constructor (`new Confirm('%s', '%s');`) to create a client-side object of type `Confirm`. This JavaScript class accompanies our mixin and implements the client-side logic to confirm that the user would indeed like to proceed with the action.

Creating the Mixin's Javascript File

Now let's create our JavaScript class to accompany our mixin. Following the convention, we'll place this file in the same package as our mixin class, such that it becomes a resource on the classpath. This is useful because when we pass a relative name to the @Import, Tapestry will start in the package of the annotated class.

Maven location: src/main/resources/net/godcode/jsclarity/mixins/confirm.js (notice this is in `src/main/resources`)

```
// A class that attaches a confirmation box (with logic)  to
// the 'onclick' event of any HTML element.
// @author Chris Lewis Apr 18, 2008 <chris@thegodcode.net>
var Confirm = Class.create();
Confirm.prototype = {
        initialize: function(element, message) {
                this.message = message;
                Event.observe($(element), 'click', this.doConfirm.bindAsEventListener(this));
        },

        doConfirm: function(e) {
                if(! confirm(this.message))
                        e.stop();
        }
}
```

Knowing a little about the prototype library will help you here, but in short, this is a JavaScript class that takes 2 arguments in its constructor: the DOM id of the HTML element (`element`) of which we will intercept `onclick` events, and the confirmation message (`message`) to display when a click is received.

That's it, we're done writing our mixin. Now let's use it!

Using our Mixin

If you launch the web application and visit the root of the context root (`/jsclarity` by default), you'll be brought to the default index page (formerly called the Start page). This is a fresh project created via the quickstart archetype which created this page for us. All we need is a PageLink or ActionLink to test our mixin with, and the index page has one. However before adding the mixin, launch the project, view the page, and make note of the 'Refresh' link. This is a simple PageLink component that links to this same page (hence, 'Refresh'). When you follow the link you'll see the page reload and the time update, so we will easily be able to see the effects of using our mixin on it. We don't need to mess with any Java code, so let's get straight to the template and add our mixin.

Maven location: src/main/webapp/Index.tml

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">
    <head>
        <title>jsclarity Start Page</title>
    </head>
    <body>
        <h1>jsclarity Start Page</h1>

        <p> This is the start page for this application, a good place to start your modifications.
            Just to prove this is live: </p>
        <p> The current time is: ${currentTime}. </p>
        <p>
            [<a t:type="pagelink" t:mixins="confirm" page="index">Refresh</a>]
        </p>
    </body>
</html>
```

Notice the addition of the `t:mixins="confirm"` parameter. Now reload the page and click on the link. Did it work? What happens if you click the 'OK' button vs the 'Cancel' button?

Suppose you want a different message to be displayed to the user. No problem. Remember that `message` parameter in our mixin class?

```
        @Parameter(value = "Are you sure?", defaultPrefix = TapestryConstants.LITERAL_BINDING_PREFIX)
        private String message;
```

This gives you total control over the message. Exploit it in the template by adding another parameter to the PageLink component:

```
        <p>
            [<a t:type="pagelink" t:mixins="confirm" t:message="Are you sure you want to delete this item?"
page="index">Delete</a>]
        </p>
```

Understanding What's Happening

At this point we've created a mixin and paired client-side functionality with it by creating a JavaScript class in an external file. In doing so we've learned how to create this pairing, make sure the script file is included in the rendered page, and how to create the client-side object from the mixin. But how does it all fit together?

Let's start with what we know:

1. Ultimately a page comes down to HTML and JavaScript.
2. Our JavaScript class is in an external file, so that file must be included.
3. Our JavaScript class uses the prototype JavaScript library, so that must be included before we include our JavaScript file.
4. Because our JavaScript class needs the DOM id of an HTML element, that element must be loaded before we instantiate an instance of our class.

Now view the source of the rendered page. It will look something like this:

```
<html>
    <head><link href="assets/tapestry/default.css" rel="stylesheet" type="text/css">
        <title>jsclarity Start Page</title>
    </head>
    <body><script src="assets/scriptaculous/prototype.js" type="text/javascript"></script><script src="assets
/scriptaculous/scriptaculous.js" type="text/javascript"></script><script src="assets/scriptaculous/effects.js"
type="text/javascript"></script><script src="assets/tapestry/tapestry.js" type="text/javascript"><
/script><script src="assets/net/godcode/jsclarity/mixins/confirm.js" type="text/javascript"></script>
        <h1>jsclarity Start Page</h1>

        <p> This is the start page for this application, a good place to start your modifications.
            Just to prove this is live: </p>

        <p> The current time is: Fri Apr 18 21:36:42 CEST 2008. </p>


        <p>
            [<a href="/JSClarity/" id="pagelink">Refresh</a>]
        </p>
    <script type="text/javascript">
<!--
Tapestry.onDOMLoaded(function() {
new Confirm('pagelink', 'Are you sure?');
});
// -->
</script></body>
</html>
```

If you scan through it carefully, you'll notice that the prototype library is included before our library, and that at the end of the document is our javascript initialization script. Also notice that our mixin filled in the missing pieces (the DOM id of the link, and the confirmation string):

```
<script type="text/javascript">
<!--
Tapestry.onDOMLoaded(function() {
new Confirm('pagelink', 'Are you sure you want to delete this item?');
});
// -->
</script>
```

Fantastic! But how did it all happen? Well, it all starts with that annotation I said we'd look at later.

@Import

This annotation is a wonderful shortcut. To find where and how Tapestry implements the magic behind it you'll need to dig into the T5 source. The beauty of course is that you don't need to understand the *how* to understand what it does.

You can apply this annotation to page, component, or mixin classes. You must provide it with either the name of a JavaScript file as a string, or an array of files. These strings represent locations in the classpath, which will be searched in a relative manner. Whenever you use this annotation, Tapestry also adds the prototype and scriptaculous libraries before the files specified in the annotation automatically. It's also smart enough not to add the same file twice. To top it off, if your files cannot be found Tapestry will fail fast by throwing an exception. To me, this is just great. I don't want my application to just suck it up and keep going. The assumption made by Tapestry here is the same for any injected asset files: that they are an integral part of the application's front end and so the breaking of the application is a logical consequence when they are missing.

JavaScriptSupport

The last missing piece is the JavaScriptSupport service. Look again at the `afterRender` method of the mixin class:

```
        @AfterRender
        public void afterRender() {
                javaScriptSupport.addScript("new Confirm('%s', '%s');", element.getClientId(), this.message));
        }
```

Now look again at the source of the rendered page in your browser:

```
<script type="text/javascript">
<!--
Tapestry.onDOMLoaded(function() {
new Confirm('pagelink', 'Are you sure you want to delete this item?');
});
// -->
</script>
```

That basically says it all. Calls to `JavaScriptSupport#addScript` result in a `<script>` block being generated at the foot of your page. If you want to see exactly what happens in the JavaScript then you'll want to look at the `tapestry.js` file, where you can find the source of `Tapestry.onDOMLoaded`. You could just take my word and know that the function argument it receives is called after the document has loaded, effectively associating the contained code with the `onload` event of the document. In our case we can know that the element to which we are attaching our JavaScript object will have loaded.

If multiple calls are made to `JavaScriptSupport#addScript`, then the scripts will be accumulated and output in a single call to `Tapestry.onDOMLoaded`. Take note if you are calling `JavaScriptSupport#addScript` from a method used to restart the render cycle. For instance, you could have a tree menu component that uses an @AfterRender method to iterate over the list of tree nodes, while `JavaScriptSupport#addScript` is used to initialize the tree. Since you only want to initialize the tree once, you may want to do something like this:

```
        @AfterRender
        private boolean phase7() {
                heartbeat.end();
                //if there is another element in the tree,
                //go to the @BeginRender method again.
                return (!iterator.hasNext());
        }

        @CleanupRender
        void phase8(MarkupWriter writer) {
                writer.end();
                String jsString = "";
                jsString += "Event.observe( window, 'load', function() {";
                jsString += " %s = new JSDragDropTree(); ";
                jsString += " %s.setTreeId('reporttree'); ";
                jsString += " %s.setImageFolder('../images/tree/'); ";
                jsString += " %s.setRenameAllowed(false); ";
                jsString += " %s.setDeleteAllowed(false); ";
                jsString += " %s.initTree(); ";
                jsString += " %s.expandAll(); ";
                jsString += " } );";
                javaScriptSupport.addScript(jsString, treeObjectName,
                                treeObjectName, treeObjectName, treeObjectName, treeObjectName,
                                treeObjectName, treeObjectName);
        }
```

Problems in a Zone

If you use this Confirm mixin on an ActionLink that has a "zone" parameter, you'll find that the ActionLink's event is triggered even if the user clicks the Cancel button. The same is true of other components that take a "zone" parameter. This is apparently because Tapestry's JavaScript-based request to update the zone occurs before this mixin's onclick JavaScript executes.

As a workaround, you can put the mixin on an *inner* element, like this:

```
        [<a t:type="actionlink" page="index">
                <span t:type="any" t:mixins="Confirm" t:message="Are you sure you want to delete this item?">
                        Delete...</a>]
                </span>
        </a>
```

(from http://tapestry-users.832.n2.nabble.com/quot-Confirm-quot-mixin-won-t-cancel-when-in-zone-td5048950.html)