1 Introduction

Splunk's SimpleXML is an XML file format to describe a custom dashboard with searches, inputs and panels. There are a number of fantastic resources[1] for building them, but I recently encountered an interesting problem. That link also discusses SplunkJS, a Javascript library that allows users to customize searches and visualizations far beyond what SimpleXML allows.

SplunkJS is usually used with raw HTML and CSS, but can be pulled into a SimpleXML file by uing the `script` attribute in the SimpleXML opening `<dashboard>` or `<form>` tag. It's easy to make a SplunkJS search and attach it to a SimpleXML visualization; it's not so easy to make a SimpleXML search and attach it to a SplunkJS visualization. This document shows you how, and shows you how to fix a peculiarity that arises from creating a well-organized ecosystem of panels and dashboards.

In later versions of Splunk, SimpleXML has a new attribute for `<panel>`, `ref`, which allows you to define a panel in a single file and drop it into a number of different dashboards without having to cut-and-paste the panel code. In the process, SimpleXML mangles the names of searches and visualizations, and so finding and manipulating those searches has become difficult.

This example uses the Splunk Unix TA (Technology Add-on), so you should download and install that. What data you use isn't really important. For our example, though, we're going to do is create a single dashboard with a single independent panel that shows the list of processes running on a host, find that panel, find it's search, find it's title, and modify the title with the name of the longest-running process.

After installing Splunk (here's the free version of the Enterprise Edition[2], limited to a half-GB of data per day) and getting it up and running, click on the App icon (the gear symbol) on the left sidebar. On the Applications list, click on "Create a New App", and provide it with a name, a directory slug, and a version number.

Now it's time to fire up your editor. We need to create three things. A dashboard, a panel, and a javascript file to perform the magic.

1.1 Literate Program

A note: this article was written with the Literate Programming[3] toolkit Noweb[4]. Where you see something that looks like ⟨*this*⟩, it's a placeholder for code described elsewhere in the document. Placeholders with an equal sign at the end of them indicate the place where that code is defined. The link (U-¿) indicates that the code you're seeing is used later in the document, and (¡-U) indicates it was used earlier but is being defined here.

[1]See URL `href="http://dev.splunk.com/getstarted`.
[2]See URL `http://www.splunk.com/en_us/download/splunk-enterprise.html`.
[3]See URL `http://en.wikipedia.org/wiki/Literate_programming`.
[4]See URL `http://www.cs.tufts.edu/~nr/noweb/`.

2 The Dashboard Files

The Dashboard file is simple. We just want to pull in a panel. This goes into `APP\_HOME/default/data/ui/views/index.xml`. Here, APP_HOME is the path to the directory slug where your app is stored. I install Splunk in `/opt` and I named my example "searchhandle," thus the path is `/opt/splunk/etc/apps/searchhandle/default/da`

⟨*index.xml*⟩≡
```
<dashboard script="title.js">
  <label>A Dashboard with a portable Panel and a Managed Title</label>
  <description>A simple demonstration integrating SimpleXML and SplunkJS</description>
  <row>
    <panel ref="cputime" />
  </row>
</dashboard>
```

The panel file is also simple. It's going to define a search and a table. It goes in `APP\_HOME/default/data/ui/panels/cputime.xml`. Note that the filename must match the `ref` attribute. I've limited the search to the last hour, just to keep from beating my poor little laptop to death.

⟨*cputime.xml*⟩≡
```
<panel>
  <table>
    <title>Long-running processes</title>
    <search id="cputimesearch">
      <query>index=os source=ps | stats latest(cpu_time) by process | sort -latest(cpu_tim
      <earliest>-1h@h</earliest>
      <latest>now</latest>
    </search>
  </table>
</panel>
```

The `<dashboard>` tag in our dashboard file has a `script` attribute. This is where we'll put our logic for manipulating the title of our panel. It's annoying that we have to put our script reference in the dashboard and not the panel. It's possible to have a file named "dashboard.js" which will be loaded for *every* XML file in your app, and then have it selectively act on panels when they appear.

3 The Javascript

Javascript files go in the `APP\_HOME/appserver/static/` directory. I've named ours `title.js`.

Splunk uses the `require` facility to import files. In the prelude to any SplunkJS interface, you must start with the `ready!` import, which doesn't allow the contents of this file to run until the Splunk MVC (Model View Controller) base library is loaded. We're also loading the `searchmanager` and two utility libraries: underscore[5] and jquery[6], both of which come with the SplunkJS UI.

The one thing we're most concerned with is the `registry`, which is a central repository where all components of the current Splunk job's client-side operations are indexed and managed.

The file's outline looks like this:

⟨*title.js*⟩≡
```
require([
    "splunkjs/ready!",
    "splunkjs/mvc/searchmanager",
    "underscore",
    "jquery"
], function(mvc, searchManager, _, $) {

    var registry = mvc.Components;

    ⟨update the title with the data⟩

    ⟨listen to the search for the data⟩

    ⟨find the search⟩

    ⟨wait for the search to be available⟩
});
```

[5]See URL `http://underscorejs.org/`.
[6]See URL `https://jquery.com/`.

In the outline, we took one of the items passed in, `mvc.Components`, and gave it a name, the *registry*. Waiting for the search to be available is as simple as listening to the registry:

⟨*wait for the search to be available*⟩≡

```
var handle = registry.on('change', findPanel);
```

Finding the search and attaching a listener to it is actually one of the two hardest parts of this code. First, because we have to *find* it, and the new panels layout makes that difficult, and secondly, because the change event mentioned above can happen multiple times, but we want to make sure we only set up our listener only once.

Below, the function `findPanel` lists through all the Splunk managed objects on the page, and finds our search. It does this by looking for a registry name that matches the ID of our search. The panel layout mangles the name, attaching the prefix "panelXX_" where XX is some arbitrary index number. (In practice, the index number is probably deterministic, but that's not useful or important if you're going to be using this panel on multiple dashboards.) Underscore's `filter` is perfect for finding out if our search is available. If it is, we disable the registry listener and proceed to the next step, sending it the search name.

⟨*find the search*⟩≡

```
var findPanel = function() {
    var panel = _.filter(registry.getInstanceNames(),
                         function(name) { return name.match(/panel\d+_cputimesearch/); });
    if (panel.length === 1) {
        registry.off('change', findPanel);
        setUpSearchListener(panel[0]);
    }
};
```

This is the most straightforward part of the code. Having found the search name, we then get the search manager, get its results manager, and then set up a listener to it that will update the title with the data.

Splunk searches manage the *task* of searching, but not the actual data. That happens in a Result, which updates regularly with the growing cache of data from the server to the browser.

This code skips a ton of details, mostly about listening to the search for failure messages. That's okay. This is just an example, and it works 99% of the time anyway. Since we're going to change the title to include the longest-running process, and our search is pre-sorted, we just need a count of one. This Result uses the same dataset as the actual visualization and puts no additional strain on the Splunk server or bandwidth between the server and the browser.

⟨*listen to the search for the data*⟩≡

```
var setUpSearchListener = function(searchname) {
    var searchmanager = registry.getInstance(searchname);
    var resultmanager = searchmanager.data("preview", {
        output_mode: "json",
        count: 1,
        offset: 0
    });
    resultmanager.on("data", updateTitle);
};
```

The last thing we do is update the title. (Remember, that's our goal). The panel's title is found in the `.panel-head h3` child DOM object. Finding the panel is trickier, but Splunk gives us an attribute with the name of the panel's filename, so jQuery can find it for us. There's a guard condition to ensure that we actually have some data to work with.

The names of the fields correspond to the final names in the search. I've always found Splunk's naming conventions to be a little fragile, but it works most of the time.

⟨*update the title with the data*⟩≡

```
    var updateTitle = function(manager, data) {
        if ( !data || !data.results || !data.results.length) {
            return;
        }

        var topprocess = data.results[0];
        $("[data-panel-ref=cputime] .panel-head h3")
            .text("Longest Running Process: " + topprocess["Process"] +
                  " (" + topprocess["CPU Time"] + ")");
    };
```

4 Navigation

One last detail: You want to be able to get to this page.

To do that, open the file at: `APP\_HOME/default/data/ui/nav/default.xml` and change the line for search to look like this:

⟨*update navigation*⟩≡
```
  <view name="index" default='true' />
  <view name="search" />
```

Now restart Splunk

And that's it. Put it all together, and you've got yourself a working application in which SplunkJS can tap into SimpleXML searches and exploit their data, even if that search is defined in an independent panel.

This code is available at my github at Splunk with SimpleXML and Javascript[7].

[7]See URL `https://github.com/elfsternberg/Splunk-SimpleXML-SplunkJS`.