An Apache Nifi Frontend Developer's Cookbook (Part 1: JS modules and bootstrapping the canvas)

This article is the first in a series of articles discussing various topics regarding frontend UI/UX development on the Apache NiFi project (versions 1.1.2+). This series will explore some of the architecture and bootstrapping of the client side JavaScript as well as various "recipes", case studies, and examples on how to accomplish certain tasks.

Introduction

Keeping up with modern web development best practices, frameworks, and anti-patterns can be intimidating. Frameworks like AngularJS, Ember.js, Knockout, Backbone.js turned the JavaScript world inside out just 3-5 years ago with their declarative styles and TDD approach. Couple that with all the ever changing ECMAscript standard, differing support of said standard in various browsers, and the fact that as memory and storage capabilities have continued to increase and at lower price points more and more functionality is expected from, and placed on, the client and you start to understand that web development is still maturing and ever changing. The tooling around web development has also added to the insanity. Polyfills to add new features/capabilities to old browsers and transpilers that offer future ECMSscript6 functionality in today's "modern" browsers. Terminology like “module bundlers vs. module loaders,” “RequireJS vs. SystemJS”, and “AMD vs. CommonJS vs. UMD vs. NativeJS(ES5 vs. ES6)”. It can quickly become overwhelming but understanding these terms is vital for web developers who desire to contribute client side code in the Apache NiFi project.

In this article we will discuss a few of the topics mentioned above and then walk through a couple of case studies complete with in depth examples and appropriate exercises. Once you have read it you should feel comfortable with:

- The modular nature of NiFi's JavaScript components
- How the canvas application is bootstrapped
- How to add events to components on NiFi's canvas
- How to add actions to NiFi's context menu

NiFi's JavaScript Modules

The Apache NiFi project strives to deliver a maintainable, testable, extensible, and reusable code base. For the client the next step in working towards these long term goals was to remove any global references to NiFi's already conceptually modular JavaScript components and then resolve any circular references. This does a couple of things

1. Provides better encapsulation, clear dependency graph, and more maintainable code
2. Enables client side unit testing (decoupled modules are easily mocked)
3. Positions NiFi to take advantage of modern JavaScript tooling like module loaders/bundlers

Previous to Apache NiFi version 1.1.2 the `nf` namespace was added to the browser's global `window` object as a container for all NiFi client modules. As each module was loaded they would add themselves to the `nf` object. Whenever a module wanted to access methods in another module:

```
//Previous to Apache NiFi version 1.1.2 the way a developer would access methods from one
nf.someModule.someFunction()
```

The issue with this approach is that there is nothing stopping developers from having two modules depend on each other and creating circular references. YUCK!

Starting with Apache NiFi 1.1.2, in order to prevent developers from inadvertently creating circular references each JavaScript module is now wrapped in an IIFE (Immediately Invoked Function Expression that runs as soon as it is defined). Once invoked, a Universal Module Definition (UMD) pattern is leveraged to detect, accept, and inject JavaScript modules loaded via **ANY** module loader (like RequireJS, SystemJS, WebPack, ..., or even script tags in the document head!). Finally, the module factory returns and the module is added to the global `nf` for storage and retrieval. The reason the UMD pattern was leveraged was because it supports all of the existing JavaScript module loader/bundlers and as of this writing the nifi-web-ui client is not utilizing any of them (RequireJS, SystemJs, WebPack, ...). Eventually a module loader may be introduced (and subsequently a module bundler to replace our custom bundler...but that is a separate effort another post!) but until then modules are still being loaded via the script tags in the document head with one major difference. **The order is strictly enforced.**

Let's consider the following example of a JavaScript module that has jquery injected into its scope:

```
// Note: `root` is the browser global `window` object.
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        //If an AMD module loader is requesting this module.
 //NOTE: The NiFi canvas application does not use an AMD module loader(see[5]).
        define(['jquery'],
            function ($) {
                return (root.nf.ModuleFactory = factory($));
            });
    } else if (typeof exports === 'object' && typeof module === 'object') {
        //If CommonJs module loader is requesting this module.
 //NOTE: The NiFi canvas application does not use an AMD module loader(see[5]).
        module.exports = (root.nf.ModuleFactory =
            factory(require('jquery')));
    } else {
        //If no module loader is being used then inject jquery from the global `$` names
 //NOTE: This is the way that the NiFi canvas application loads JS modules
        root.nf.ModuleFactory = factory(root.$);
    }
}(this, function ($) {
    //Use the injected jquery '$'
    var body = $('body');
    // private method
    function notHelloOrGoodbye(){
        //Use the injected jquery
        var body = $('body');
    };
    // public methods (see below)
    function hi(){};
    function bye(){};
    // Exposed public methods
    return {
        hello: hi,
        goodbye: bye
    };
}));
```

Now, the experienced JavaScript developer may notice:

1. There is a lot of boilerplate code.
2. Ultimately the module was added to the global `nf` so how is this approach any different than accessing JavaScript modules from the global scope?

Each module needs the module(s) it depends on to be injected into its scope but NiFi JavaScript modules are reusable and need to be agnostic to the module loading strategy being used. Module loaders like RequireJS and SystemJS provide a type of cache which allows developers to store and retrieve JavaScript modules and UMD supports that use case. What about the case when a developer simply wants to load a NiFi JavaScript module with a script tag? It turns out that our `nf` namespace (remember the `nf` object?)

is already acting as a type of cache and so by leveraging the UMD pattern and existing browser capabilities we are able to 'roll our own' JavaScript module loader in the case that our NiFi modules are being **NOT** being loaded with a module loader.

Bootstrapping the NiFi canvas application

The nf-canvas-bootstrap.js module is the last module loaded but the first piece of code that runs when the NiFi canvas starts up. It waits for the DOM ready event to be fired, then it configures and bootstraps the angular application (more on that in a follow up article), and then is responsible for initializing the rest of the modules (many of which are dynamically manipulating the DOM) . During this startup phase control of a few Javascript modules are inverted. As a matter of convenience the injected modules are listed below:

Module	**Injected with a reference to:**
nf-canvas-utils.js	• nf-graph.js • nf-birdseye.js • nf-actions.js • nf-snippet.js • nf-canvas.js [1]
nf-quickselect.js	• nf-actions.js
nf-shell.js	• nf-context-menu.js
nf-component-version.js	• nf-settings.js
nf-connection-configuration.js	• nf-birdseye.js • nf-graph.js
nf-controller-service.js	• nf-controller-services.js
nf-reporting-task.js	• nf-settings.js
nf-context-menu.js	• nf-actions.js

[Table 1]

Ok...but why?

There are four main strategical advantages gained by leveraging the UMD pattern, decoupling circular references, and leveraging inversion of control during the NiFi bootstrapping process:

1. Module loading order is enforced preventing future circular references.
2. With the previous circular references and global variables creating test mocks was difficult at best. Now, client side test mocks and unit testing will be straight forward.

3. The NiFi build process was ahead of its time. It leverages maven to bundle all the JavaScript modules into a single file, minify it, and gzip compress it in order to minimize the number of http requests (as well as the size of the files requested) necessary to load the application. Today there are many, feature rich, JavaScript module loaders and bundlers that offer built in dependency management, caching (module storage and retrieval), bundling of resources (complete with dead code removal), etc. These changes to the the NiFi JavaScript modules opens up the possibility of removing some of the custom maven build processes and allows the introduction of a modern module loader/bundler in order to better align with industry best practices and to remove the burden of learning and maintaining a custom build process that web developers will not be familiar with.
4. NiFi JavaScript modules are reusable and agnostic to the module loading strategy being used. **[5]** Cool huh?

Case Study: Adding event listeners to components on the canvas

In this section we will explore the nf-context-menu.js **[2]** module and how the "right-click" event listener is bound to each component displayed on the NiFi canvas. Below are links to each of the NiFi canvas component types available:

- nf-processor.js
- nf-funnel.js
- nf-label.js
- nf-process-group.js
- nf-remote-process-group.js
- nf-port.js

Each of these NiFi canvas component type modules are responsible for several things:

- Rendering the components type on the canvas
- Maintaining a map of each rendered instance of their respective component types (e.g. the nf-label.js module maintains a map of all rendered instances of a label component type.)
- Persisting each rendered component position
- (...there are few other responsibilities that I will leave to the reader to investigate for themselves)

However, for this case study there are a couple responsibilities that deserve a special mention:

1. Inverting control of any injected modules and maintaining references to them (see example). This process takes place during the initialization of the nf-graph.js module **[3]** which is responsible for injecting the nf-context.menu.js module reference into the appropriate NiFi canvas components.
2. Attaching event listeners to each rendered instance of the respective component type. Let's examine how a 'contextmenu.selection' **[4]** listener is attached to a NiFi canvas label component and how the NiFi canvas context menu is requested:

   Example: NiFi canvas 'Label' component:

   The nf-graph.js module inverts control of the nf-context-menu.js module to the nf-label.js module during its initialization (which, consequently, is the last module initialized in the bootstrapping process). When the nf-label.js module renders a 'Label' component it leverages the D3 3.x library to attach a listener to that particular component for the 'contextmenu.selection' event.

   Exercise 1:

   One of the Apache NiFi community members recently submitted a PR to add a 'quick select' capability to components on the canvas that listens for a 'dblclick' event on each component and then opens the configuration dialog for that component. This is a great example of how to implement event listeners and I encourage you to review the PR here and then attempt to add your own.

Case Study: Adding actions to the context menu

At this point we have a good understanding of how the NiFi canvas context menu is requested for a single component on the canvas but what about when multiple components on the canvas are selected? Let's examine how a requested NiFi canvas context menu determines what menu items are available and how the nf-context-menu.js module is able to invoke actions in the application:

> **Example: How NiFi context menu items are determined based on components selected on the canvas:**
>
> No matter how many components on the canvas are selected the context menu is actually requested by the component on the canvas that is right clicked. However, the list of actions available in the requested context menu are based on the selected components. So, how do we get the collection of selected components? Well, good news, the nf-canvas-utils.js module is capable of returning a list of the currently selected components on the canvas. Ok, so now that we know how to get the selection where is it determined which menu items to include in the context menu. The way this process works is pretty straight forward. Each menu item is an object in a list. Most of the properties of a menu item object are for things like the display text and the displayed icon for a context menu item. However, there are two properties that deserve some special attention:
>
> 1. The `condition` function. When the show() method of the context menu is called the list of ALL possible menu item objects are processed and each condition tested and if it passes then the item is added to the menu.
> 2. The `menuItem` object's `action` property. Notice that this property is a string. Also notice (see **Table [1]**) that control of the nf-actions.js module has been inverted to the nf.context.menu.js module. Since JavaScript provides access to public properties on an object via bracket notation (see all property accessors here) what this `action` property string really represents is any public property of the nf-actions.js module (you can find all available public nf-action.js properties on this object here).
>
> **Exercise 2:**
>
> Add a new menu item to the context menu. (Hint: you will need to reuse an existing private condition function from the nf-context-meu.js module or you will need to add a new one. You will also need to reuse an existing publicly available nf-actions.js module function or you will need to add a new one.)

Conclusion

Whew! That was a lot to cover and a lot of the information here is pretty dense. I hope you found this informative and helpful in your quest to contribute to the client side code for Apache NiFi. I look forward to addressing any questions and thanks for reading!

NOTES:

[1] The nf-canvas.js should not be injected in any other module except for nf-canvas-utils.js and any direct interaction with the nf-canvas.js should be avoided. Use the nf-canvas-utils.js module to interface with the nf-canvas.js module.

[2] The nf-context-menu.js, as you may suspect, is responsible for rendering and maintaining the available menu options listed for the rendered NiFi canvas context menu.

[3] The nf-graph.js module depends on and acts as an interface to of each of the NiFi canvas component type modules throughout the NiFi canvas application.

[4] The D3 JavaScript Library is used extensively throughout the NiFi canvas application. This article does not focus on best practices nor does it offer any in depth explanation on how D3 is implemented or its expected behavior. A nice blog discussing D3 and context menus can be found here.

[5] If a developer is building a custom processor UI and wants to use RequireJs to load nf-common.js that functionality is there. If some other developer wants to use SystemJS to load nf-common.js that functionality is supported by the JavaScript modules provided by Apache NiFi.