Why AMD

Multi-version support of RequireJS

# FWRequireJS: AMD-style modules in Adobe Fireworks

As browser-based applications have become more complex, JavaScript developers have worked out various approaches to including multiple modules on the page without polluting the global namespace. Complex Adobe Fireworks extensions can benefit from similar solutions, and I’ve tried a few different approaches in the past. What FWRequireJS offers is [James Burke’s](6) excellent, battle-tested [RequireJS](1) module loader along with some additional infrastructure that permits it to work in the Fireworks JS environment.

The RequireJS site offers more information about the [Asynchronous Module Definition](2) standard and why it’s useful.

## Installing FWRequireJS

If you use git, you can just clone the [FWRequireJS GitHub repository](4). If not, you can download a [.zip file](5) containing all of the files.

Once you have the repository, there are only two files you need from it: `fwrequire.js` and `require.js`. `require.js` is an unmodified copy of the 1.0.8 release of [RequireJS](3). `fwrequire.js` wraps up the RequireJS code and enables it to run within Fireworks.

# Using FWRequireJS

A Fireworks extension will typically contain a number of .jsf files that provide related functionality. Perhaps you’re writing an extension that exports the currently selected elements, and you want to have one command in the \*Commands\* menu that exports the elements on a transparent background, one that exports them on white, one that exports them on black, etc.

You might find yourself copying the same code to each of those .jsf files and just tweaking it slightly to change, say, the background color value. But a better approach would be to separate out the common code into a module that can then be loaded by the .jsf files. That way, you can modify your library in just one place, rather than having to make a change in each .jsf file.

Let’s say the directory containing your extension files looks like this:

Commands/

My Export Commands/

Export Selection on Black.jsf

Export Selection on Transparent.jsf

Export Selection on White.jsf

To add FWRequireJS support, create a `lib/` sub-directory, and put the `require.js` and `fwrequire.js` files in it. You can also create an `export.js` file there that will hold your reusable module code. The directory should now look like this:

Commands/

My Export Commands/

lib/

fwrequire.js

require.js

export.js

Export Selection on Black.jsf

Export Selection on Transparent.jsf

Export Selection on White.jsf

## Defining modules

Now that the files are set up, you can create a module using the global `define()` function. This global is instantiated by some boilerplate code in the .jsf files, which we’ll cover later. For now, we can assume the FWRequireJS library has been loaded and `define()` is ready to be used.

The `define()` call can take a number of parameters, but in its simplest form, you just pass it a function. The return value of that function should be the module that is defined by the .js file. Each .js file should define only one module.

Your `export.js` file may look something like this:

define(function() {

// vars and functions that are local to the module

// can be defined here

// the function returns an object containing the public

// methods implemented by the module

return {

exportSelection: function(backgroundColor)

{

...

}

};

});

The `exportSelection()` method will be called from the .jsf files to do the actual exporting, and each file can pass in a different `backgroundColor` parameter. See the RequireJS site for [more examples](7) of defining modules.

## Requiring modules

The .jsf file gets access to the module defined in `export.js` by calling a global `require()` function. Before it can do so, however, it must make sure the FWRequireJS library is loaded. To do this, you must include a couple lines of boilerplate code at the beginning of every .jsf file that makes use of the FWRequireJS library:

if (typeof require != "function" || !require.version)

fw.runScript(fw.currentScriptDir + "/lib/fwrequire.js");

This if-statement checks that there’s a global function called `require` and that it has a `version` property. If neither of these is true, then it loads `fwrequire.js` in a `lib/` sub-directory, which will, in turn, load `require.js` from the same directory. By supplying some configuration settings, you can store the files in a different directory, but FWRequireJS will look in `lib/` by default.

Why does \*every\* .jsf file that uses FWRequireJS need this code? Why can’t you just load it in the first one? Well, unlike a webpage, you have no way of controlling which .jsf files are run or in which order. That’s up to the user interacting with the \*Commands\* menu in Fireworks. So \*any\* .jsf file might be the first to be run, and therefore \*every\* file that uses it has to check for, and possibly load, FWRequireJS. See the [Multi-multi-version support](#multi-multi-version-support) section for more details.

Once the FWRequireJS library has been loaded, requiring a module is straightforward:

require([

"export"

], function(

exportModule)

{

exportModule.exportSelection("#000000");

});

The first parameter to `require()` is usually an array of one or more strings that name the modules that this file depends on. Once those modules have been loaded, the second parameter to `require()` will be called back with references to them.

The module names in the dependencies array are mapped to file paths that are relative to the directory where `fwrequire.js` was loaded. So in the example above, `"export"` would be loaded from `lib/export.js`. If you have a module located in a sub-directory of `lib/`, like `files.js`:

Commands/

My Export Commands/

lib/

fwrequire.js

require.js

export.js

utils/

files.js

Export Selection on Black.jsf

...

then that module can be loaded with code that looks like this:

require([

"export",

"utils/files"

], function(

exportModule,

files)

{

exportModule.exportSelection("#000000", files.getName());

});

Just remember that module paths are relative to the directory from which you loaded `fwrequire.js`, \*not\* the directory containing the .jsf file that’s using `require()`. This root directory can also be specified via the `baseUrl` property of a configuration object passed to `require()`. See the [configuration]() section for details.

The name of the parameter that’s mapped to the loaded library can be whatever you like, and doesn’t have to be exactly the same as the module file name. In the example above, the `export` module is called `exportModule` in the callback function, since `export` is a reserved word in Fireworks JavaScript. Just make sure that the order of the parameters is exactly the same as the order of strings in the dependencies array.

Module filenames should end in .js, not .jsf, because you don’t want them to show up in the \*Commands\* menu. These files provide functionality to other .jsf files, not to the end-user.

Note that although the AMD specification has “asynchronous” right there in the name, files are always loaded synchronously in the Fireworks environment. And since Fireworks doesn’t support any HTTP request functionality, all modules must be loaded from local files.

## Configuring FWRequireJS

requirePath and fwrequirePath have to be absolute paths

can include the filename

## FWRequireJS vs. RequireJS

Some of the power RequireJS provides is only useful in the browser. For instance, the ability to build all of the scripts that are needed for a project into a single minified file is great for speeding webpage load times. But that isn’t necessary in Fireworks, where all of the files can be loaded synchronously from the file system. In the documentation, you can ignore the references to `<script>` tags and attributes like `data-main`.

One difference with RequireJS is that in FWRequireJS, module names that end in .js are always loaded relative to the current root directory. In RequireJS, modules ending in .js are loaded relative to the HTML page, but since there’s no concept of a page in Fireworks, FWRequireJS will look for all modules relative to the directory containing `fwrequire.js`.

## Multi-multi-version support

One of the nice features of RequireJS is that it makes it easy to load different versions of the same library on the same page. If the different versions are installed at different paths, a given call to `require()` can pick either or both versions of the library by specifying the appropriate relative path.

### The problem

This capability is even more crucial in an environment like Fireworks, which presents several unique problems that need to be solved by a module loader:

\* Every script runs in a single shared global scope, and any script can stomp on any other’s globals.

\* The global scope persists until the application is closed.

\* Extension developers cannot automatically load scripts when Fireworks starts up.

\* Developers have no control over which extensions are installed or in which order users run them, which is very different than the challenges faced by a team building a single site with a well-defined and stable production environment.

\* There is no event system or threading. After the user runs a .jsf file from the \*Commands\* menu, the script simply runs to completion, possibly running other scripts along the way.

Since developers have no way of controlling which extensions a user installs, there’s no way to guarantee which version of `require()` is loaded in the global scope. Even among a single developer’s extensions, a user may have installed extensions `A.jsf` and `B.jsf`, where A uses RequireJS 1.0 and B uses RequireJS 2.0 (which isn’t released yet, but just to illustrate the example).

If the user runs command A first, then the global `require()` function will be version 1.0. If the user then runs command B, that command will be stuck with version 1.0 of RequireJS, which may not be compatible. The reverse scenario poses the same problems. Of course, scripts could always check which version of `require()` is loaded and then reload their preferred version. But if the user is going back and forth between running command A and then command B, that’s a lot of unnecessary loading and unloading.

### The solution

FWRequireJS addresses this problem by essentially sandboxing each extension that uses `require()`, allowing each one to have its own copy of RequireJS. Different extensions can then load their preferred version of RequireJS without interfering with each other. Of course, the modules that each instance of `require()` loads will also be kept separate.

When it’s run, FWRequireJS creates global `require()` and `define()` functions (they’re actually the same function). This function isn’t the same `require()` that’s defined by RequireJS. Rather, its job is to load the appropriate copy of RequireJS and dispatch the call to that copy.

An example will hopefully make things a little more concrete. Let’s say you have two extensions installed, each using slightly different versions of RequireJS, as well as different modules:

Commands/

Extension 1/

lib/

fwrequire.js

require.js

files.js

Command A.jsf

Command B.jsf

Extension 2/

lib/

fwrequire.js

require.js

files.js

export.js

Command C.jsf

Command D.jsf

After launching Fireworks, the user runs \*Command D\* first. That .jsf file checks for a global `require()`, doesn’t find it, and then runs the `fwrequire.js` in its `lib/` directory.

After FWRequireJS is loaded, the `Command D.jsf` script calls `require()` to load the `files` and `export` modules. This global function looks at the root module directory for this command, `Commands/Extension 2/lib` in this case, and checks whether it already has loaded `require.js` from that path. It hasn’t, so it saves off the current values of `require` and `define`, and then runs `Commands/Extension 2/lib/require.js`.

Loading `require.js` will redefine the `require()` and `define()` globals with the actual functions from RequireJS. The original `require()` from FWRequireJS (confused yet?) is still executing, however, and after loading RequireJS it calls `require()` with the arguments it originally received. At this point, RequireJS takes over and does its module-loading magic.

After RequireJS is done, execution switches back to FWRequireJS, which saves off the `require` and `define` globals and restores its original copies. So at this point, the global `require()` points to the function defined by `Extension 2/lib/fwrequire.js`, which is keeping track of the RequireJS globals created by `Extension 2/lib/require.js`.

Now the user runs \*Command A\*. The .jsf checks for a global `require()` function and finds it in this case, so it doesn’t need to run `Extension 1/lib/fwrequire.js`. But the FWRequireJS version of `require()` sees that the call is using `Extension 1/lib/` as the root directory, and that the version of `require.js` in that directory hasn’t been loaded yet. So it does the same thing as before: saves off the globals, runs `require.js`, dispatches the call to the new instance of `require()`, and then restores the globals when that call has finished.

All of this may seem like a lot of unnecessary machinery just for making your code more modular. And that’s probably true if all you’re doing is writing a few simple extensions or don’t mind creating big, monolithic scripts. But if you start to create many extensions and want to share code between them, the FWRequireJS loader can make the process a lot more efficient.

[1]: http://www.requirejs.org/

[2]: http://www.requirejs.org/docs/whyamd.html

[3]: <https://github.com/jrburke/requirejs>

[4]: <https://github.com/fwextensions/fwrequirejs>

[5]: https://github.com/fwextensions/fwrequirejs/zipball/master

[6]: https://github.com/jrburke

[7]: http://www.requirejs.org/docs/api.html#define

`fwrequire.js` creates a global `require()` function that dispatches calls to the real RequireJS function depending on which script file is calling it. This lets each extension install an independent copy of RequireJS and its modules. The global dispatcher figures out the path to the command that’s calling it, then passes control to a `Context` object that manages the RequireJS instance at that path. The context will save off the current `require()` function, load its own instances of RequireJS, then pass the arguments to that `require()`. Once the execution is finished, it restores the original fwrequire version of `require()`.

the first fwrequire.js is called twice, once to set up global require dispatcher

then to instantiate a Context

calls to require() from different paths will then call the fwrequire.js in

that path

implementations of the Context that dispatches calls to require modules can

be kept separate

only the dispatcher is shared across all contexts

files that define modules should never be called directly with runScript

need to check for the existence of require, and then call your fwrequire.js

if it's not there

# The problem

As browser-based applications have become more complex, JavaScript developers have worked out various approaches to including multiple modules on the page without polluting the global namespace.

# The solution

The `Context` is separated from the dispatcher

Adds about 30ms the first time the library is loaded, then a few ms of overhead for each `require()` call

The dispatcher expects its registerContext to be called with a context object, which should have an execute() method

The context should make use of the config object returned by registerContext