Why AMD

Multi-version support of RequireJS

# Multi-multi-version support

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

requirePath and fwrequirePath have to be absolute paths

can include the filename

# 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.

Fireworks has several unique problems that need to be addressed 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.

RequireJS provides a great AMD-compatible loading mechanism that works well even outside a browser.

But 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.

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.

# The solution

`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 `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