Notes on the Infusion Module Loader

General motivation

Working with multiple git/npm artifacts in the context of the GPII is already becoming unmanageable - even without any of the further modularisation proposed in Notes on Modularisation of Infusion.

For example, we have a dependency structure where every artifact depends on grunt-gpii via grunt itself, and there are multiple paths to dependency on infusion, directly via universal, and indirectly via kettle and node-jqUnit. All of this as well as creating huge redundancy (since all of these dependencies should really be shared) strongly obstructs development - since, for example, making a change in one of these deeply nested projects can require dozens of git commit, push, npm install steps (sometimes recursively if updating, for example, universal's dependency in order to reach it from windows or linux).

We need a module loader which, in addition, heads off problems such as GPII-492 (duplicate infusion modules) at the source - we waste a lot of time scanning npm for dependencies which should really appear just once.

Module sharing is, unfortunately, deeply anti-religious in the current development climate. Here is a interesting thread on grunt-cli where it is explained why a "shared grunt" is not supported - whilst several more mature participants to the thread point out that this completely frustrates the possibility of scalable development.

Our proposal for a module loader is unambitious in a few ways - since we are happy with the npm packaging infrastructure and the basic operation of the registry as well as the module loading process for individual modules. For example, npm install <github url> will directly and unproblematically install a single dependency for a single project - although it will, unhelpfully, recursively install all of its nested dependencies also. require <physical file path> will direct node to load a particular module from a filesystem path - without invoking any of its problematic recursive resolution rules. We plan to defer to the registry and physical module loading for individual module loads - it is only the overall orchestration that will be reimplemented.

Earlier thinking

Earliest thinking on this subject is at http://issues.gpii.net/browse/GPII-3 which describes a requirement for a modified file layout, which is currently impossible due to the hard-coded inter-project file paths used in several places between the repositories (universal, linux and windows).

More recent thinking is at "Issues with the current module loader" at http://wiki.gpii.net/w/Architecture_Ideas,_Sketches,_and_Meeting_Notes

Thinking and related work of others

As mentioned earlier, the grunt-cli issue thread Installing both grunt-cli and grunt globally is an excellent jumping-off point to understand the current development climate. Whilst numerous developers with clear experience and hard scalability requirements do pitch in, their concerns are mostly ignored by the core team on ideological grounds. The line is that "this is the node.js model, you should follow it". Interestingly one of the more experienced contributors does drive a wedge in this argument, by pointing out that "node itself is more enterprise ready than grunt".

Refers to "use a global grunt install if no local install found". Report is "Grunt will probably never defer to globally installed Grunt as the last resort" and refers to the previous thread holding "reasons why global Grunt is bad" whereas in fact it just holds rhetoric.

An interesting page referenced from both of these discussions is Building Command Line Tools in Node with Liftoff which acknowledges at least one use case for global dependencies, namely CLI tools. It points out various hard limitations in node's handling of global modules as installed by npm -

1. Globally installed modules cannot `require` other globally installed modules.
2. Locally installed modules cannot `require` globally installed modules.

This shows how "half-assed" this support for global installation really is. The mess is compounded by the fact that npm's location for global modules and CLI tools is subtly different from node's support, operated by the require.paths property - node docs Loading from the global folders explains that, for example, it will look in locations such as $HOME/.node_modules. The 3rd option, $PREFIX/lib/node, closely resembles the location used by npm, but does not actually agree with it - see npm's documentation on npm folders. This explains Global installs on Unix systems go to `{prefix}/lib/node_modules`. Global installs on Windows go to `{prefix}/node_modules` (that is, no `lib` folder).

npm link

One of the core use cases we mention above, speeding development when a deeply nested, multiply shared dependency is being worked on, is handled by an npm function known as **`npm link`** - the following blog posting on the node.js blog (6/4/2011) explains how this feature came about and was properly implemented for npm 1.0: link. The thinking does appear to be about a very closely related problem - however, there are two serious issues:

- No support is possible for this feature under windows (except through the prohibitive route of compiling your node.exe under Cygwin) since the feature relies on symlinks
- The installation of the modules is indeed global - the developer is guided into a globally stateful workflow and must remember to "unlink" their package otherwise all work will be corrupted in all packages. Ridiculously, this still needs to be done manually - see npm unlink does not unlink from global path

The shell of a plan

The things which are shared, and the things which are not shared, need to be considered very carefully. Our use case is clearly wider than the one acknowledged by "liftoff" - it is not just CLI tools themselves, but the entire build and load chain, which needs to be able to establish agreement on what modules are loadable from where. A core use case is from grunt plugins themselves, and from our replacement for npm - let's call it **`ipm`**. We need to identify the barest minimum of the "module bootstrap loader" which receives this treatment. It clearly makes no sense to put all of infusion, or all of any sizable project, into a shared module area - but nonetheless there should be something in that area, which quickly and authoritatively determines, given any invocation point, which the particular instance of infusion is that should service the current project, and ensure that there is exactly one of it.

We need the concept of "workspaces".... ("multi-projects", etc. or some other hackneyed and disagreeable term), which is, a "space" of related projects which have been checked out/installed and have agreed to cooperate on consolidating dependencies and module resolution. This space of projects will be guaranteed to have exactly one copy of infusion (and any other cooperating dependencies) resolvable within it, and in addition, to have never even attempted to check out a duplicate version of such a dependency. Tasks like "npm dedupe" and the constant harping about how disk space and network bandwidth are cheap notwithstanding, npm installs are now extremely slow and wasteful and only getting more so. A checkout of express 4.0 includes 7 copies of "bluebird" and 2 separate instances of all of selenium in its 80Mb project tree. "lodash" may be very small, but guaranteeing to have one copy of it in every project due to grunt depending on it and grunt insisting on being separately installed in every npm module is a route to insanity. The idea of "locally installed tools" makes sense but we need much more control over what the scope of "local" actually is - it can't simply be "the smallest unit that our package manager knows how to manage".

Such a workspace will be delimited in the filesystem by a file named "ipmRoot.json" (say) held at its root. This will serve a variety of functions -

- Providing clearer delimitation of the space searched through for modules - the "keep looking upwards for node_modules" rule is easily prone to accidental leakage
- Providing a portable and easily readable place for dumping "redirect rules" of the kind morally operated by **`npm link`** - this makes it easy to continue, for example, working with a given checkout of a project (say, one's "main checkout of infusion" just in a particular space, whilst leaving other spaces unaffected

We can defer to **`require <physical path>`** for our physical mechanism for loading a particular module within node.js - however, we can't defer to **`npm install <git URL, etc.>`** for our mechanism for installing an individual dependency of one package - because npm will still continue to cascade endlessly through derived dependencies of that package before we can stop it. The package.json packaging scheme however is good, as is the registry itself, and these are both pieces of infrastructure we can easily reuse with good profit.

In theory we would like to be able to ensure that any of our packages can still be "npm install"ed without error (only inefficiency) but this will require much more care and thought. This is the kind of issue that requires us to somehow interact with one or possibly both of the schemes listed above for truly global loading. We need to eliminate the requirement for the grunt dedupe-infusion step to ensure "in time" that npm installs continue to be valid "in many cases" - where these cases include a single infusion-aware dependency installed as a dependency of a non-infusion-aware host project. Should the host project end up installing more than one infusion-aware dependency it's reasonable to expect they should start buying into our module loading infrastructure. Conversely, we will still defer to vanilla npm install to install leaf dependencies of our own projects - although where these become "well-known" (such as when.js) we would want to consolidate them too. This can be done by simply allocating them a well-known name in our global namespace.

fluid.require needs to be beefed up so that

1. fluid's global is just straightforwardly advertised as fluid.global - allowing access to all such shared dependencies to be bootstrapped from any participating library
2. it stores a registry of module loaders associated with each (npm) project name. fluid.resolveModulePath ("universal/src/testData") etc. should then be capable of getting filesystem paths into any such dependency for the purpose of resolving resources. cf ancient art in the form of ClassLoader. getResourceAsStream().

For point 2 - this hopefully eliminates the need for "kettleModuleLoader.js" - since we will just use the utility to rebase file paths rather than needing to export "require".

However, the actual use of fluid.require will become unnecessary - except for gaining access to "fluid" itself - since the entire purpose of the module loader is to cause all dependencies to be located and installed automatically. As we have been saying for a while, we will initially go with a "**load everything loadable**" model, and then over time we can think about more sophisticated schemes for only loading files holding defaults blocks and/or global functions that have actually been referenced. "load everything loadable" needs to be qualified by platform and/or other environmental dependencies though. This militates towards using a separate marker file (**`ipmPackage.json`**?) rather than stuffing things into package.json - which has only a few hard-wired kinds of env dependencies such as os, cpu, etc.

The "bare minimum module loader" will be small

- Firstly, to ensure it represents as stable as possible commitment to something which is globally shared
- Secondly, to minimise its impact should it end up unfortunately being installed multiple times locally

We would hope that it just consists of a couple of dozen lines of code.

How will we know which module loader to use to install a dependency?

In general, we won't. So we will use the lightweight process from "ipm" to install the raw image of each dependency in the first instance. Then we will inspect its installed packaging to see if it includes a certain marker (it seems custom fields are permitted in package.json - or else a separate file). If the marker is not there, we will run genuine npm install on it. If it is there, we will continue to cascade using ipm.