Notes on Kettle

(notes from Community Meeting held on 16/07/14)

VOTT is KETTLE?

Kettle has historically been our server-side JavaScript environment, matching Infusion on the client-side.
The Kettle we have today is in fact the 3rd thing of that name:

i) "Old Old Kettle" was built using Mozilla's Rhino environment on the JVM for Fluid Engage during 2008

ii) "Old Kettle" was a brief and ill-fated attempt to port "Old Old Kettle" one for one onto node.js - this is housed in github here: https://github.com/fluid-project/old-kettle

These "Old Kettles" used the strategy of managing a "reasonably credible" simulation of the DOM on the serverside in order to run the copy of jQuery on which Infusion relies. During work on "Old Kettle" we found that whilst this could be done, it led to an extremely unstable technology stack and had few architectural advantages. The advantages we hoped to get included the ability to appear to issue XHR requests on the server, and thus use most of the same code for I/O on the server as we did on the client - including all of the courtesies offered by jQuery's .ajax() wrappers - rather than getting involved in the nuts and bolts of node.js I/O.
[In practice, in "New Kettle" we abstract over I/O by means of the "dataSource" API which also existed in a similar incarnation in "Old Kettles". The incomplete nature of this API is one of the main architectural plagues of New Kettle just as it was in the Old]

iii) "New Kettle" arose from a module named "GPII requests" during the very earliest phase of Cloud4All development in mid-2011. This emerged initially during the GPII Hackathon held at Boulder and was elevated into its own npm module later in the year. It is held in github here: https://github.com/fluid-project/kettle

PRIMARY DISTINCTIONS BETWEEN "OLD KETTLES" and "NEW KETTLE"

a) fidelity of simulation of the jQuery and window/browser environment for supported code

b) "New Kettle" relies for all primary functionality on the standard "express" node module rather than implementing all of this functionality in the raw (operating I/O, handling HTTP requests, handling routing, parsing and serialising of request headers and bodies)

Bootstrapping and the primordial process for Infusion within node.js and "New Kettle"

"New Kettle" uses an "extremely thin" simulation of jQuery and the underlying "window" object which was directly ripped off from the jQuery codebase into a file named jquery.standalone.js held in GitHub here: https://github.com/fluid-project/infusion/blob/master/src/framework/core/js/jquery.standalone.js
This copies the bare minimum of the core utilities from jQuery which Infusion relies on. Infusion's node.js environment and context is boostrapped from the Infusion node module loader which is housed at https://github.com/fluid-project/infusion/blob/master/src/module. Most of the included contents in this fake window object are actually for QUnit rather than for the section of jquery.standalone itself. We need to take this route because it is easier to "go the whole hog" and convince QUnit that it is really (mostly) in a browser than to somehow back out of the decision we already made to expose jQuery to it.
In https://github.com/fluid-project/infusion/blob/master/src/module/fluid.js you can see the fabrication of the "primordial context and window complex" which occurs at line https://github.com/fluid-project/infusion/blob/master/src/module/fluid.js#L28 - in practice this creates a very small circularly linked structure which is sufficient to get all observers to agree on the browser axiom that "the global object is the same as the object named window".
This "fluid.js" is what bootstraps the process of loading "Infusion as a node module" which starts whenever anyone issues

```
require("infusion")
```

from a requesting node.js module.
For example, "New Kettle's" bootstrap begins on this line:
https://github.com/fluid-project/kettle/blob/master/kettle.js#L13
After Infusion itself has bootstrapped into node, it is generally not appropriate to make further direct use of node.js module loading semantics. Infusion's semantic is different in that it relies on the existence of a stably named global namespace, corresponding to Infusion's own "global" object (the one set up in fluid.js). This happens for free in the browser, but requires special setting up in node.js.

MODULE LOADING IN KETTLE and/or INFUSION IN NODE.JS

For this purpose we make use of fluid.require , or else a "loader" exported by fluid.js by the API fluid.getLoader. These ensure that, as well as being loaded as a module, the loaded code is properly contextualised with the "single well-known global object" held by Infusion.
Here's an example of the kind of thing that can go wrong if we don't have facilities for rebasing directories via allowing module loaders to be exported from one module to the other:
Existing hard-coded file references between repositories:
https://github.com/GPII/windows/blob/master/gpii.js#L24
A worse example here:
https://github.com/GPII/windows/blob/master/tests/acceptanceTests/AcceptanceTests_include.js#L28

THE MYSTERIOUS "kettleModuleLoader.js" file

Every Kettle-aware node module must embed a file named kettleModuleLoader.js at its root, with these contents:

```
// The purpose of this file is to be copied to a location for which the
// require function is needed. It allows to find node modules relative
// to that location that are otherwise non-resolvable.
module.exports = require;
```

Kettle Organisation:

The "basic unit" of construction of Kettle applications is an Infusion component known as "kettle.app". These are grouped into higher-level groupings known as a "kettle.server".
Kettle applications are bootstrapped in purely declarative terms based on pure JSON files named "configs". These contain pure JSON encodings of the configuration which in Infusion we issue to fluid.defaults.
Here is an example from within the GPII:
https://github.com/GPII/universal/blob/master/gpii/node_modules/flowManager/configs/development.json
This has a section at the base named "includes" which is extra to the options material sent to fluid.defaults. This contains a list of files relative to the current path which contain the JavaScript code necessary to operate the component defined in the options. It is because i) these filenames are relative to the current module, and ii) the process of loading these files is operated by the framework (kettle) rather than by any code actually within the current module that the file kettleModuleLoader.js is required.
In this case this config resolves hierarchically to a further config:
https://github.com/GPII/universal/blob/master/gpii/node_modules/flowManager/configs/base.json
Which in turn includes a reference to "modules" which must be loaded via the node module loader relative to the current directory.
The root "config" for one configuration of the entire GPII infrastructure is held here:
https://github.com/GPII/universal/blob/master/gpii/configs/fm.ps.sr.dr.mm.os.lms.development.json
This configuration codes for the top-level overall "server" which corresponds to "an entire thing listening on a port" which within express corresponds to "an express server". You can see this being constructed in Kettle's server.js at https://github.com/fluid-project/kettle/blob/master/lib/server.js#L182
Despite "New Kettle" being layered on top of Express, we've tried to avoid the explicit appearance of express primitives within the configuration of Kettle apps. However, there has unavoidably been some leakage of express semantics into the higher levels - for example, the string format used to express express's routing rules appears within the configuration of Kettle handlers - an example is at:
https://github.com/GPII/universal/blob/master/gpii/node_modules/flowManager/src/FlowManager.js#L82
Properly this should occur inside one of the "configs" representing the FlowManager but a lot of this material still unnecessarily appears in code.
The format of the string "/user/:token/login" directly corresponds to express's routing rules documented at http://expressjs.com/2x/guide.html#routing
(Note that the version of express we use is pretty old and hasn't been moved in a while (currently at "express": "~3.4.3",)
In order to deal with asynchrony within Kettle we make use of a core promises library known as "when.js". You will see these promises used in various Kettle request handlers in order to slightly sanitise the process of issuing and receiving multiple asynchronous operations. In practice this doesn't really sanitise them all that much - here is an example from our matchmaker component:
https://github.com/GPII/universal/blob/master/gpii/node_modules/matchMaker/src/MatchMaker.js#L169
When's API is documented at https://github.com/cujojs/when/blob/master/docs/api.md - we are currently several major versions behind its master version of 3.x

DATASOURCES AND URLS WITHIN KETTLE

The ultimate purpose of the "dataSource" abstraction is to abstract over whether particular functionality is hosted locally or remotely. Unfortunately we haven't delivered on very much of this goal within either old or new Kettle yet - but the system we have now relies on a particular concrete implementation known as a "URL DataSource" - this implementation is held in Kettle at
https://github.com/fluid-project/kettle/blob/master/lib/dataSource.js#L224
The GPII components which need to adapt to being hosted locally (as part of the same server) or hosted in a distributed way (amongst a group of servers) have their interaction mediated by this dataSource API. This API is as follows:

```
dataSource.get(directModel, callback) - reads material held at the coordinates "directModel"
```

```
dataSource.set(directModel, model, callback) - writes material "model" to the coordinates "directModel"
```

The "direct model" can be seen as a JSON summary of the contents of a URL. It expresses an "index" into some set of state which can be read and written.
In theory the dataSource API shouldn't intrude directly into component configuration but in practice it, and the machinery of resolving URLs, is still exposed at top level.
For example, here are some dataSources attached to the flowManager:
https://github.com/GPII/universal/blob/master/gpii/node_modules/flowManager/src/FlowManager.js#L42
In the development configuration, everything is hosted on the same server - so these are configured to point to our own port on localhost via the following configuration:
https://github.com/GPII/universal/blob/master/gpii/node_modules/flowManager/configs/development.json#L14
This is also not what should happen. In the local configuration, we should not communicate to ourselves via HTTP at all, but should instead abridge the dataSource contract to a simple function call to one of our own components. We currently don't have the implementation or extra component metadata required to make this work, and it is a source of fragility and poor performance.
In a distributed configuration, real URLs are distributed down to these dataSources via IoCSS held in this configuration:
https://github.com/GPII/universal/blob/master/gpii/node_modules/flowManager/configs/cloudBased.json - for example, to the real preferences server held at http://preferences.gpii.net/user/%token
This appeals to the following grade -
https://github.com/GPII/universal/blob/master/gpii/node_modules/flowManager/src/FlowManager.js#L139
This should probably also be held in a "config" by itself. This configures away the deviceReporter and lifecycleManager which are not hosted in this configuration, and exposes a new endpoint called "settings".
At the moment configuration is split haphazardly between "config" files and fluid.defaults blocks, and we should make more effort to consolidate them in the former. .js files should only contain bare functions.
We need to improve the "dataSource" abstraction so that

i) it doesn't pollute the base grades with the URL configuration
ii) doesn't push the requirement for the "callbackWrapper" onto the author of each Kettle app
iii) allows direct abridgment to function calls in the case of a local endpoint, rather than roundtripping through HTTP over loopback.