Coding and Commit Standards

Fluid Code Standards

Fluid's coding standards represent guidelines that we mutually follow and respect as a community. They're not hard and fast rules. They are intended to help make our work easier to follow, more transparent, and accessible to those outside the community. Fluid committers are expected to follow these conventions, to help review others' code and encourage adoption of the conventions, and to help refine these guidelines over time.

Code Commits

JIRA and github

All work should be associated with a particular JIRA issue. Fluid's github repositories are integrated with our JIRA issue tracker, allowing us to easily cross-reference specific code commits with issues in JIRA. This is accomplished by the inclusion of a JIRA issue number in front of every commit log. JIRA can then automatically associate the change with the bug, so that watchers of an issue can stay apprised of its progress. Pull requests should also be named with the JIRA issue number at the front, to make identifying pull request in github easier.

JIRA issues should be meaningful and describe the task, bug, or feature in a way that can be understood by the community. Opaque or general descriptions should be avoided. If you have a large task that will involve a number of substantial commits, consider breaking it up into subtasks.

(see: JIRA Best Practices)

Commit Logs

All commits logs should include, at minimum, the following information:

1. A reference to the JIRA issue this commit applies to (at the beginning of the first line)
2. A short and meaningful summary of the commit, on the first line
3. A meaningful commit log describing the contents of the change

In rare cases, a commit may be trivial or entirely cosmetic (code reformatting, fixing typos in comments, etc). In those cases, it is acceptable to use the NOJIRA prefix for your log. Even in those cases, a meaningful summary and descriptive commit message should be included.

Example of a Good Commit Message

FLUID-99999: Add foo function to whirlygig

Refactored the whirlygig to include a new foo algorithm based on the wireframes provided by the designers. (http://linkToWireFrames)

Pushing to the Project Repo During Bug Parade and Code Freeze Periods

The Fluid release process involves two phases: bug parade and code freeze. Bug parade involves a concentrated period of community-wide bug squashing. We focus on a set of the highest-priority open issues that affect components in the upcoming release. These inevitably include, but are not limited to critical and blocker bugs.
Code freeze provides a period of time for us to test and verify the codebase immediately before release. Pushes to the project repo are ordinarily prohibited, except for a small subset required for the process of releasing.

Pushes to your own fork/clone of the project repo and to project repos that aren't currently in a release process are exempt from the above restrictions. Although it is preferable that you are involved to assist the community in the release efforts.

**During Bug Parade**

1. You may only push commits to the project repo which are for issues that have been included on Bug Parade
2. Bugs can be added to the bug parade at any time, upon approval by the QA Lead and/or Release Manager and with a notification to the community (via list or irc channel).
   - Anyone can nominate a bug to be added to the bug parade
3. All commits must undergo code review before an issue can be closed, and before the end of bug parade
   - The review process consists of a code review and testing of the commit
   - The result of the code review should be indicated on the jira
   - All community members are encouraged to lend a hand with code reviews, but each commit must also be reviewed by a committer.

**During code freeze**

Ordinarily, no commits should pushed to the project repo during code freeze. Exceptions include:

1. Commits which affect comments, licenses, and README files
2. Changes that are required as part of the process of cutting a release:
   - Updating version numbers
   - Tagging the release
3. Blocker bugs that are found during release testing, and have been identified as a necessary fix before release
   - The bug should be discussed with the QA lead prior to being made a blocker
   - Fixes must be submitted as a patch (attached to the jira) or as a pull request (made through github and linked to on the jira), and under go code review before being pushed to the project repo.
   - All commits must under go testing before the issue can be closed

**Using Branches**

When working with forks/clones of Fluid repositories, all branches should be named after the JIRA issue they are related to. This makes it easy for the community to track your progress and know when a branch is complete.

Branches should be deleted when they have been merged back into the project repo or have been abandoned.

(see: [GIT Tips and Tricks](#))

**Code Conventions**

As Fluid grows and our code is used by a wider community of users, we need to keep code quality high and ensure the integrity of our builds. Our daily builds are also used as a showcase of our work, so pushes to the project repo should not break the build. If you find you've pushed broken code into the project repo, please fix it expediently or back your code out so that the rest of the community is not affected by a broken build.

All code that is pushed into the project repository should be easily forked/cloned and built by the community. It shouldn't be hard-wired to your particular development environment, and should include build scripts or quick instructions on how to get the code working.

Experimental or exploratory code should be placed in your own public repo (e.g. your personal github space), where the expectations about quality can be more lax to allow experimentation and innovation. You are still encouraged to create meaningful JIRA issues and avoid the use of NOJIRA where possible. This is especially the case if the intention is to promote this code to the project repo.

Code that is pushed into the project repo should not include debug statements (e.g. console.log). If you require debug statements in your code you should push it to your own public repo (e.g. your personal github space).

**Unit Tests**

Production-level code needs to be accompanied by a reasonable suite of unit tests. This helps others confirm that the code is working as intended, and allows the community to adopt more agile refactoring techniques while being more confident in our ability to avoid regressions. More information about unit testing is available on the [Writing JavaScript Unit Tests](#) page.

**JavaScript**

**ESLint**

JavaScript is a highly dynamic and loose language, and many common errors are not picked up until run time. In order to avoid errors and common pitfalls in the language, all code should be regularly checked using ESLint. Configuration for ESLint is controlled through eslintrc and eslintignore files found in the repo.

**Style**
Coding style is always a contentious issue, often leading to unproductive "bike shed" debates. Style is a very personal decision. At the same time, style conventions help to make a code base more readable and approachable. In order to avoid arguments about coding style, we've chosen to follow someone else's coding style. In the case of JavaScript code, we're using Douglas Crockford's code conventions. They're not perfect, and we don't follow them slavishly, but they provide us with a baseline for easily readable, consistent code that is less error prone. Use them.

For hints on configuring Aptana/Eclipse for some of Douglas Crockford's code conventions see Aptana (Eclipse) settings for coding conventions

CSS Naming Conventions

Infusion adheres to a strict naming convention for class name selectors. The template for classnames (i.e. class names used for styling) looks like fl-[fluid:multiWordThing]-[fluid:multiWordRole]-[fluid:state], with some guidelines:

1. multiple words are in camelCase, so they might look like fl-[fluid:multiWordThing]-[fluid:multiWordRole]-[fluid:state]
2. components are always the first "thing", so they would look like fl-[fluid:componentName]-[fluid:thing]-[fluid:role]-[fluid:state]

[fluid:thing] = **required** the concept the class name is referring to at the most general yet still meaningful level (eg. fl-tabs, fl-widget, fl-col, fl-container, etc)
[fluid:role] = **optional** the purpose or action of THING, which could be a more detailed version of THING (eg. fl-tabs-centered, fl-col-flex, fl-container-500, fl-widget-titleBar)
[fluid:state] = **optional** a modifier of the ROLE or THING, which is only temporary and dependant on other actions (eg. fl-button-left-disabled, fl-widget-content-draggable)

Some examples of class names:

- fl-col-flex: a flexible width column
- fl-tabs-left: left-oriented tabs
- fl-grid-caption: the caption of an image in a grid
- fl-widget-titlebar: the titlebar of a widget

The template for Infusion selectors (i.e.classnames used for programmatic manipulation of the DOM) looks like flc-[fluid:componentName]-[fluid:thing]-[fluid:role]-[fluid:state], with the same guidelines as selectors.

Some examples of component selectors:

- flc-progress-label: the label for a progress bar
- flc-reorderer-dropWarning: the drop warning used with the reorderer
- flc-inlineEdit-textEditButton: the 'edit' button used with an inline edit
- flc-pager-pageLink-skip: a page link that should be skipped

When combining class names for styling and class names for selectors, the convention is

```
<div class="flc-* fl-*">
```

so that the DOM selectors are before the CSS selectors.

The reason we do this is to separate the different purposes of selectors: one to style things, one to find things in the DOM. This way, you could change your behavioural code without affecting your appearance and vice versa.