The Javascript Specification

March 29, 2009

Contents

1 Introduction

The Javascript frontend has been redesigned to be more modular, and, hopefully, somewhat simpler. Instead of consisting of a single large object definition in a single (and huge) file, it is now split up into smaller method definitions in separate files. It is highly suggested that you read through at least this section and the next, Initialization.

At the highest level, the program is composed of three "things." These three "things" are the `fb_hash` function/object, the `fb.Interface` instance/class, and the feedback classes.

1.1 `fb_hash`

The most prominent of these is the `fb_hash` function/object. To reduce the chance of namespace collisions, the old `fb` object has been renamed to `fb_hash`, where the "hash" part

will eventually be an actual random string (i.e., `fb_234lkh6ui`). Everything is written inside of anonymous functional closures, which effectively give this object its original name of `fb`, so from now own, unless necessary, `fb_hash` will be referred to as `fb`. Also, to maintain some sort of parallelism in the code, the `fb` object itself is the old `fb.init()` function (everything else is a class, where the variable is actually the constructor, so that has been mimicked here). As such, in order to invoke the program, all that is needed is to call `fb_hash()`. Finally, everything else is a property or a method of `fb` (i.e., `fb` is the namespace in which the entire program resides).

1.2 `fb.Interface`

The second main "thing" is the `fb.Interface` instance/class. The `fb.Interface` class encapsulates and creates the user interface aspect of the program. Its internals will be discussed later, but the important thing to know is that there is only one instance of this class, instantiated as the program is initialized (given that the user is authorized) and stored at `fb.i`. Also, everything with regards to layout, display, formatting, etc., is dealt with by the `fb.Interface` class. Everything else is dealt with by the feedback classes.

1.3 Feedback classes

The last main "thing" in the program is actually many "things": the feedback classes. For each type of feedback allowed (e.g., page-comments, changing text, changing colors, changing images, etc.) there is a class (stored as a method of the `fb` object). For example, the `fb.Comment` class encapsulates page-comments. It stores all of the state relating to them, does all of the page-comment processing that is necessary, and has all of the page-comment CRUD methods (with the exception that some of the CRUD work is done in the respective feedback interface classes, see `fb.Interface` below).

1.4 Miscellaneous

Finally, a few other side notes. jQuery is still located at `fb.$`, and `fb.env` still exists to hold that state which is not contained in some feedback class (like the `init` boolean, the `current_page` string, the `url_token`, etc.). However, `fb.env.logged_in` has been renamed to `fb.env.authorized` to match the backend and to help generalize the program (when giving feedback on a page accepting public feedback, you are still authorized even though you are not logged in). Also, two new variables have been introduced, `fb.env.get_address` and `fb.env.post_address`. These are the URLs to which we submit the HTTP GET and POST requests, respectively. Finally, everything is URI encoded (by `encodeURI()`) before being submitted. This changes nothing in the backend, but in the frontend, upon instantiation, the feedback classes need to decode the necessary fields (by `decodeURI()`).

2 Initialization

2.1 File Structure

As mentioned at the beginning, the original program has been rewritten to be more modular. One part of that is the use of classes defined in the `fb` namespace, and the other part is the use of multiple files. At this point, the program consists of the following files:

1. `fb_hash.js`
2. `fb.jQuery.js`
3. `fb.Common.js`
4. `fb.Interface.js`
5. `fb.Interface.comment.js`
6. `fb.Comment.js`

The first of these files defines `fb_hash()`, and does nothing else. The second loads the jQuery library and the jQuery windowName Transporter plugin under the `fb.$` namespace. The third defines miscellaneous helper functions (see `fb.Common.js` below). The fourth defines the `fb.Interface` class (see `fb.Interface` below). The fifth is a sample feedback interface class, and the sixth is a sample feedback class. In general, the order of the files should be as follows:

1. `fb_hash.js`
2. `fb.jQuery.js`
3. `fb.Common.js`
4. `fb.Interface.js`
5. All feedback interface class definition files (one per class)
6. All feedback class definition files (one per class)

To generate the single Javascript include file a new Rails controller has been introduced: `FeedbackjsController`. This controller currently has one action, `index`, that concatenates these files in order (appending the `fb_hash()` call at the end) and then renders the result as a Javascript file. In addition, it caches the result so that the file need not be re-generated on every call. To expire the cache, run `rake jscache:clear` in the `coreapp` directory. Note that the cache needs to be expired manually after every edit of anything in `coreapp/app/js`. Finally, because of this new controller, the Javascript frontend should now be loaded from

`http://localhost:3000/feedbackjs`.

2.2 Initialization

(Largely from Arthur's e-mail) Initialization begins with loading Javascript file rendered by `FeedbackjsController:index`. The first part of this file (`fb_hash.js`) defines the function `fb_hash()`, which is the namespace within which the rest of the program resides. This gives us a reference to something, and we extend this reference with numerous attributes in the files in `coreapp/app/js` via the

```
(function(fb) {
  // var $ = fb.$;
  ...
})(fb_hash);
```

pattern. This pattern defines an anonymous function with one argument (`fb`), and then immediately calls it with `fb_hash` as the argument. Inside of this function, any variables can be used, given they are declared first (using the Javascript keyword `var`). For example, the first line can be uncommented to reassign the jQuery variable as `$`, allowing the use of `$` as usual. Furthermore, within this function, we have access to the `fb_hash` object by the name `fb,` and can extend `fb_hash` via `fb.` These extensions are available globally because `fb_hash` was defined (globally) at the very beginning. Thus, when `fb_hash()` is called and executed at the very end (as a function), all of its attributes (e.g., `fb.env`, `fb.Comment`, etc.) are already there.

After the `fb_hash` object is ready (i.e., all of its attributes are defined), we call `fb_hash()` to begin initializing the program. `fb_hash()` sets up the environment, pulls out `current_page` and `url_token` (which is the empty string if it is not found neither in the URL nor in a cookie), sets `get_address` and `post_address`, and executes the first request for comments. The callback for the request (defined anonymously inline) checks authorization and, conditioned on authorization, builds the program interface and displays the received comments. Checking authorization includes setting the `fb.env.authorized` boolean and stopping when necessary. Building the program interface is accomplished by instantiating the only instance of `fb.Interface`, `fb.i`, and displaying the received comments is accomplished by calling `fb.Comment.get_callback` with the received data and the boolean `true` to signal the rendering of all new comments).

Note that, first, as mentioned before, `fb.i` is the one and only instance of `fb.Interface`. Trying to create another instance will result in an exception being thrown. Also note that the interface must be initialized before we try to build any comments. Building comments requires that the respective inner class of `fb.Interface` already be defined (more on this later), so building a comment before the interface has been set up will result in errors.

3 fb.Interface

As mentioned before, the `fb.Interface` class encapsulates everything that has to do with the user interface aspect of the program. As such, the only state it stores has to do with the interface; all other state is stored elsewhere. Also, since an instance of `fb.Interface` is an instance of the user interface, having multiple instances does not make sense. Trying to instantiate more than one instance of `fb.Interface` will cause an exception to be thrown.

3.1 fb.Interface()

Upon instantiation, the `fb.Interface` class does three things. First, it should pull in any stylesheets that the interface needs. As this may take some time to complete, this should be the very first thing the constructor does. (The mock-up does not use stylesheets, but their inclusion would present no difficulty.) Second, it generates the interface of the program. In the mock-up, this means creating a main "window" (`div`) in which the comment interface can reside. Third, it instantiates all of the feedback interfaces classes with itself as argument. This allows the feedback interface classes to access the properties and methods of `fb.Interface`. The new instances are then stored as instance variables of `fb.Interface`. This pattern is used because it guarantees four important things:

1. The interface, along with the individual feedback interfaces, is not initialized if the user is not authorized
2. The individual feedback interfaces are initialized after the main interface is initialized
3. The individual feedback interfaces are initialized before any instance of their respective feedback class
4. There is only one instance of every feedback interface.

The first point is taken care of in `fb_hash()`, as the program exits before initializing the interface if the user is not authorized. The second point is taken care of as the feedback interfaces are initialized at the end of the `fb.Interface` constructor, meaning that the rest of the interface has already been completely initialized. The third point is taken care of as we know that by the time the `fb.Interface` constructor completes, all of the individual feedback interfaces have been completely initialized. Since the first instances of the feedback classes are instantiated after the `fb.Interface` constructor is called, we are guaranteed that by the time these instances are built, the individual feedback interfaces are already initialized. Also, note that the third point is a requirement as the constructors of the feedback classes attempt to build their instance of feedback (see below (3.2)). Finally, the fourth point is taken care of as we know that the constructor of the `fb.Interface` class is only called once, and that is the only place where instances of the feedback interface classes are made.

3.2 Feedback interface classes

As mentioned before, there is a class for every type of feedback. These classes will be discussed later, but for now we need to know that they are expected to have at least the instance methods `render` and `remove`, and call a method `build` in the constructor. Returning to the `fb.Interface` class, `fb.Interface` has an "inner" class for each feedback class, where the "inner" classes are now called "feedback interface classes." As an example, for page-comments, the `fb.Comment` class encapsulates the actual comments and `fb.Interface.comment` holds all of the interface methods for comments. Within the feedback interface classes, the properties and methods of the `fb.Interface` class can be accessed via `self` if the pattern given in `fb.Interface.comment.js` is followed. In addition to following this pattern, there are other requirements of the feedback interface classes.

3.2.1 Constructor

The constructor of each feedback interface class should initialize the interface for its associated type of feedback, as well as define the appropriate methods for itself.

3.2.2 Methods

Each feedback interface class should have at least three methods:

`build` Creates (and returns) whatever needs to be created for a new instance of that type of feedback (e.g., the DOM representation of a comment)

`render` Renders an instance of a type of feedback (e.g., inserting the DOM element returned by `build` into the DOM or making something visible)

`remove` Removes an instance of a type of feedback from the interface

As mentioned before, `build` is called in the constructor of a feedback class (e.g., the constructor for the `fb.Comment` class calls `fb.i.comment.build(this)`). `render` and `remove`, however, are only called through their respective methods in their associated feedback class. In order to have complete encapsulation by each feedback class of their respective type of feedback, each feedback class also has the instance methods `render` and `remove`. Note that it is definitely more natural to render a comment (`Comment.render()`) than it is to tell the comment part of the interface to render a comment instance (`fb.i.comment.render(Comment)`). To implement this, the `render` method of the feedback class should call the respective `render` method in `fb.Interface`, and the `remove` method of the feedback class should call the respective `remove` method in `fb.Interface`. However, it is expected that the `render` method of the feedback class does nothing else, while the `remove` method completes the rest of its destructive functionality after calling the `fb.Interface remove` method.

4 The feedback classes

As mentioned several times before, there is a feedback class for each type of feedback. Each feedback class encapsulates a single type of feedback. For example, the `fb.Comment` class encapsulates the page-comment type of feedback. As with the feedback interface classes, there are also requirements placed on the feedback classes.

4.1 Constructor

The constructor of each feedback class should do at least the following:

- Call the build method in the respective feedback interface class, storing the result in an instance variable `build`
- Add each new instance to the class variable `all` (described below)
- Add each new instance to the class variable `unrendered` (described below)

4.2 Properties and Methods

Each feedback class should have at least two class variables:

`all` An associative array mapping `feedback_id`s to instances of the feedback class. Should hold every instance of its respective feedback class

`unrendered` An associative array of form similar to `all` that hold every unrendered instance of its respective feedback class

Note that building and rendering are two separate operations, and so a `unrendered` array is necessary.

Every feedback class should also support the instance methods `remove` and `render`, and the class (or static) methods `get`, `post`, and `render`. The instance method `render`, as discussed above in 3.2, should do nothing but call the `render` method of the respective feedback interface class with `this` as argument. As rendering is a purely interface-related task, it should be clear that the instance method `render` should do nothing else. The instance method `remove`, on the other hand, should call the `remove` method of the respective feedback interface class, and then complete the rest of its destructive functionality. As a method, it should completely remove the instance of feedback from the program (including removing the instance from the `all` and `unrendered` array, and destroying all instance variables). The class methods `get` and `post` should perform their respective actions for their associated type of feedback, with `post` assuming that it is being called in the context of a form with the appropriate fields for the associated type of feedback (e.g., `fb.Comment.post()` assumes that it is being called in the cotenxt of a form with the fields `content` and `target`, where "in the context of a form" means that the method's `this` variable is a reference to a form). The class method `render` should merely call the `render` method of every element in the `unrendered` array.

5 fb.Common.js

`fb.Common.js` contains all general helper methods, where are methods are defined under the `fb` namespace (as opposed to some being defined as plugins and extensions to jQuery). If needed/wanted, jQuery can also be extended (see the end of `fb.Common.js`), but to avoid confusion, all helper methods should be defined under the `fb` namespace. However, there is an exception. Methods that are inherently related to the interface should be defined as instance methods, by extending the prototype method, of the `fb.Interface` class. For example, the old `fb.div()` function has been moved to `fb.i.div()` as an instance method of the `fb.Interface` class. On the other hand, `assert`, which is just a general helper function, is defined in `fb.Common.js` as `fb.assert()`.

6 Testing

Arthur has already implemented an integration test as a `rake` task using (Fire)Watir. More specific unit tests of the Javascript could also be written using QUnit. It would be trivial to

create another action of `FeedbackjsController` that would generate Javascript for testing (e.g., it could leave out the `fb_hash()` call at the end and replace it with the suite of unit tests). After finalizing the design of the interface aspect of the Javascript, this should probably be the first task.

7 Next Ideas

This is a list of possible changes to the Javascript to either make simpler or generalize better. If implemented, they will be moved into an appropriate section of this document.

- The structure of the feedback classes is definitely in question. As it is now, they are separate entities with no connection, and if more are added, getting the new feedback would require one HTTP GET for each feedback class. This is far from optimal. One possible fix is to make each feedback class inherit (or "inherit") from a feedback "superclass." This "superclass" can do the delegation of work on the response to a get, allowing one HTTP GET to suffice for updating all types of feedback. Also, if this schema will be similar to how the Rails backend will end up, the parallelism would be nice.

- Right now, when the Javascript frontend sends a request for the feedbacks to the Rails backend, the reply contains all feedbacks, including those the frontend already has. In addition, the Javascript frontend has to do a lot of processing to determine which feedbacks are new, and whether or not any have been deleted. The overhead of sending redundant data and doing data processing in Javascript could be avoided by changing the format of the request for feedbacks and the response. If the Javascript frontend tells the Rails app what feedbacks it already has, then the Rails app can send only the feedbacks the frontend does not have plus a list of the `feedback_id`s of the feedbacks that have been deleted. This would significantly reduce the size of the response as well as the amount of time (and space) spent processing the response in the frontend. The downside, however, is the possibility of the frontend having to send a huge list of `feedback_id`s. This could be solved by using HTTP POST for the request instead of HTTP GET. This may be an abuse of POST's intended uses, but it would also solve the problem of having to send a `current_page` parameter that is too long.