Javascript Coding Standards

Overview

RP's standards document that describes platform, testing, formatting, and best practices for developing applications in javascript. The objectives for these standards are based on the following:

- provide an environment to produce stable, cross-browser web applications
- provide a well established framework to enable team development for both server and client side operations
- establish rules that promote "readable" code enforcing easily recognizable patterns
- promote testable and well tested code

Standards are based on the following references:

Javascript Platform Roadmap

There is a slide show that describes new enhancements and provide a roadmap that describes our javascript development goals. The document is here.

Standards References:

- Writing Testable Javascript, Shane Tomlinson

- Expert Javascript, Mark Daggett
- Test-Driven JavaScript Development, Christian Johansen
- JavaScrip Ninja, John Resig/Bear Bibeault
- Maintainable Javascript & Best of Fluent Video, Nicholas Zakas
- Building iPhone Apps With HTML, CSS and Javascript, Johathan Stark
- Introducing HTML5, Bruce Lawson
- Javascript Patterns, Stoyan Stefanov
- Javascript: The good parts, Douglas Crockford (http://javascript.crockford.com)

Other Resources:

- Coders at Work, Peter Seibel
- jQuery Cookbook, Cody Lindley
- jQuery Pocket Reference, David Flanagan
- Videos from Yahoo theater especially the multi-part series from Douglas Crockford
- Roundpeg's MVC JavaScript Platform Slide Deck
- For Javascript/CSS Browser Compatibility see can i use.
- Javascript Code Complexity, JSComplexity

ECMA Script 5

Every attempt should be made to comply with the ECMA 5 standard and be aware of it's capabilities. At a minimum, 'use strict' should be placed at the top of all function/classes. JSHint will warn if 'use strict' is not found.

Development Platform

Javascript IDE

Our preferred javascript IDE is WebStorm. We have a roundpeg.xml configuration file to enforce our formatting standard. For Mac users the file goes in ~/Library/Preferences/WebIde50/codestyles. If WebStorm isn't used, we encourage Eclipse with the javascript plugin, Sublime or Mate with appropriate plugins. Any IDE will work but using WebStorm with our configuration settings is the easiest way to create format-compliant code. Using an alternate IDE may require a post processor to help enforce standards.

Web Container

Web applications that RP develops are targeted for Restful web containers. Development of javascript based web applications requires Tomcat 7.x. (A slightly lighter weight alternative is jetty, but any J2EE complaint container will work.)

Most JavaScript projects make use of JSP servlets, so you may not see any .html files. JSPs are created to enable server side processing and configuration and provide separation between production and test driven source.

Some projects use node as the server and have a single index.html file. These projects use 'head' as a component loader.

(see javascript development environment document for installation and configuration instructions)

Logging

Logging for test, development and production debugging is a big part of RP's JavaScript development. RP has developed (and is continuing to develop) a Remote Logging lib/utility similar to log4j that is used to log to the local console (Safari, Chrome, FF5, & IE9) as well as to a remote service listener. Developers are encouraged to add log and assert statements to help debug and validate code.

Examples of typical use:

- log.debug('event handler: ', event.type, ', data: ', event.data);
- log.info('initialize service');
- log.warn('this should not happen here...');
- log.error('exception caught', error);
- log.assert((some condition that should always be true));
- log.toString(obj); // to dump the objects name/value pairs

Some legacy code include logs that are initialized near the top of a class like this:

```
var MyClass = function() {
    var clz = this,
  log = RemoteLogger.createLogger('MyClass');
};
```

The preferred way is to have the ApplicationFactory create logs for each class. So you would see this:

```
var MyClass = function(options) {
    var clz = this,
  log = options.log;

 // more code...

 // constructor tests
 if (!log) throw new Error('MyClass must be constructed with a valid log object');
};
```

MVC Package Layout

All RP javascript projects use a standard MVC layout with components, controllers, models, views, services, events, and delegates. Testing packages use a similar layout but add mocks and data packages. A single index page (index.jsp/index.html/index.aspx) accesses ApplicationFactory to construct application components based on a set of options that inject values and components that are mapped from their DOM counterparts.

Below is an example of a typical project.

```
project/
 - app/
  - assets/
  - components/
  - controllers/
  - delegates/
  - events/
  - models/
  - services/
  - views/
  - index.html
 - build/
  - application.js
  - application.css
  - index.html
  - app.cache
  - favicon.ico
 - includes.js
 - config/
  - Config.js
  - site-config.json
 - libs/
 - test/
  - components/
  - controllers/
  - delegates/
  - events/
  - models/
  - services/
 - mocha/
 - Gruntfile.js
 - package.json
```

Components

Roundpeg is slowly building a library of standard components such as buttons, drop-downs etc. Components are instantiated from controllers and has a standard method createComponent(parent). All components have full unit tests. Components never reference services but have callback hooks to controllers to request data when needed. Controllers populate data directly to components.

An alternative to creating component classes is to create/use a builder class, usually called ComponentBuilderDelegate. This can be used to build all pages, buttons, panels, tables, etc. for small applications or can be combined with component classes, usually page containers to create larger applications. These components usually rely somewhat on Bootstrap.css.

Building an entire application using components creates a very extensible platform. Pages, views, containers, dialogs and low level components can be specified in configuration and built dynamically as the application loads. But, the downside is that this approach requires a specialized skill set foreign to most HTML/CSS coders. A good alternative to this is to use templates as combined with lodash (underscore) ejs (see below).

Controllers

At a minimum, each project two controllers: 1) ApplicationFactory, and 2) ViewController. The application factory creates delegates, services and other controllers in its initialize() method. The application factory also creates the central event bus, or "dispatcher". The view controller coordinates actions between services and UI views and actions.

Delegates

Delegates are the worker classes. Workers include parsers, calculators, formatters, validators, etc. Delegates are also created to simply offload work from controllers, data access or service objects. Below is an example of a typical validator delegate. As you can see, this validator actually uses another validation delegate to access a library of validation methods.

```
var ValidationDelegate = function(options) {
    'use strict';
    var delegate = this,
  log = options.log,
  validator = options.commonValidator;

 this.validateCountry = function(country, errors) {
  log.info('validate the country object');

  if (!errors) {
   errors = [];
  }

  validator.validateLength( 'countryCode:', country.countryCode, 3, 10, errors );
        validator.validateLength( 'countryName:', country.countryName, 1, 50, errors
);

  return errors;
 };

 // constructor validation
    if (!log) throw new Error("validation delegate must be constructed with a log");
 if (!validator) throw new Error("validation delegate must be constructed with a
validation object");
};
```

Events

The events package includes all custom event types used throughout the system. It is supported by dispatcher class (EventDispatcher or the newer CentralDispatcher) used as the system's central event bus. Event classes should include static constructor methods as well as static event type definitions. Here is a typical example:

```
var ApplicationStateEvent = function(type, data) {
    'use strict';
    this.type = type || ApplicationReadyEvent.READY;
    this.data = data;
};

ApplicationStateEvent.APPLICATION_READY = 'ApplicationStateEvent_ApplicationReady';

ApplicationStateEvent.createReadyEvent = function(obj) {
    return new ApplicationStateEvent( ApplicationStateEvent.APPLICATION_READY, obj );
};
```

All event handler methods should end with the key word "Handler" and their function parameter should always be "event". This removes any ambiguity or question that this method is invoked by some triggered event, either from the central dispatcher or from a support system (ajax, button clicks, etc).

Callbacks

Callbacks follow the node standard signature of callback(err, result). The naming convention for the callback function is either the word "callback", or somethingCallback, e.g., completeCallback or updateCallback. In any case, the word callback should be at the end of the name.

Callbacks should be defined as methods like this:

```
var myfunction = function(id, callback) {
    var customer,
  callback;

 foundCallback = function(err, result) {
  if (err) return callback( err, null );

  customer = result;
  // do some stuff with the customer...

  return callback( err, customer );
 };

 dao.findCustomerById( connection, id, callback );
};
```

Models

Data Models

All data models are stored in the models package. Models are usually simple value objects but sometimes have additional functionality. Below is a typical example of a value object. The constructor accepts a params argument that acts as a copy-constructor. Some of the values are set to defaults. All value members are public. Here are two examples of the data model pattern.

```
var UserQuestion = function(params) {
 'use strict';
 if (!params) params = {};

 this.token = params.token;
 this.name = params.name;
 this.email = params.email;
 this.topic = params.topic;
 this.text = params.text
 this.questionSource = params.questionSource || 'web';
};
```

Request Models

Each service request must have an associated request model used for query, find, and save. These models inherit from abstract classes to insure that the proper userSessionToken accompanies each request. See ProfMed for examples.

Services

The services package is were all remote connections are made, usually through Ajax or WebSocket communications. Success and error handling methods are implemented a single class, CommonDataService. Request objects, parsing delegates, and specific data events are required. Application factory configures each data service with the appropriate parser, event class, logger etc. Events are typically listened to and handled in view controllers.

Service methods should always accept a single request object with a "createParams()" method to pull prepared parameters. There are two abstract request classes, AbstractQueryRequest and AbstractUpdateRequest that should be extended to provide a common interface. Request objects are created in the models package and should have associated tests.

Common Data Service

All data services use CommonDataService to pull standard json response data from remote servers. This class has a number of supporting classes including request models, parse delegate and an event class with event types to support success and failure for the three implemented methods:

- query: given a concrete implementation of AbstractQueryRequest, query returns a page-able list of domain objects. Objects are parsed by accessing parseList() in the domain parser
- save: given a domain class and AbstractUpdateRequest, post the domain class to the server for update or insert. A successful return is always a single data model parsed by parseModel()
- find: a find request object populated with the domain object's id is sent to the server. Successful response is parsed by parseModel()

Note: *There is currently a bug in jquery that attempts to cache results in some browsers. As a work-around, each request is appended with* `UID.create( '/verb?req=' )` *where the verb is the resource's verb, e.g., query, save, find, etc.*

Implementing a Data Service

A full implementation of a standard domain service layer will create about 11 files and require modifications to test/index.jsp, app/includes.jsp, ApplicationFactory.js, and ApplicationFactoryTests.js. This does not include integration into dependent or new view controllers. You will use the following Abstract and base classes:

- CommonDataService - configured in ApplicationFactory
- AbstractModelParser - extend and implement parseModel and parseList to return response.data to CommonDataService
- AbstractQueryRequest - extend and implement/override createParams()
- AbstractUpdateRequest - extend and implement/override createParams()

To speed the process of creating this code, you should use the CreateServiceLayer code generation tool.

Views

Zero, Or Near-Zero HTML Projects

The views package usually contains a single file. There is a single main container file (MainContainer.jsp) that acts as a container for all visual components. It is referenced from the index.jsp as an include inside the body tag. This file may have additional includes to pull in forms and alternate views. A single method AppUI.bindComponents() is defined in MainContainer.jsp that contains all jQuery selector to object logic. This class is the only place where selectors should be used.

Template Based Projects

Alternative to the zero-html approach is to allow HTML/CSS coders to create templates with "backing' javascript. This works especially well with the full suite of jquery and bootstrap libraries. Javascript is configured through view controllers and complex methods (closures) are attached the the backing view code to enable the view to access and update data, change views, and send alerts–all through delegated closures orchestrated by the parent controller. In general terms, the HTML template code is read in from a "templates" folder and supplied all the data that it needs to build a view. Backing code uses jquery to bind listeners to specific components in the view and communicates with the controller with direct calls. Backing code does not require unit tests because all complex methods are actually defined and tested in controller or delegate code.

Assets

The assets folder is where css, images, icons, etc. are stored. The assets folder is ignored for unit tests. Most projects use less to compile css.

Including Javascript Files

We currently use two approaches to includes 1) script tags from an included file, usually includes.jsp, and 2) a dynamic loader based on javascript usually called includes.js. The reason for a separate file is to made the included files available to both the primary application and the test framework.

The 'head' library is used with includes.js to load all javascript in parallel. This greatly speeds application load, at least during development. Tests are also declared in includes.js but are only loaded when BootStrap (defined in the index file) is set to BootStrap.includeTests = true. (See IPC Food Cost Calculator project for an example).

Unit and Integration Testing

Getting Started

Before you can run unit tests effectively you will need to install grunt and grunt-cli. See this getting-started page for help...

Legacy unit testing is implemented using QUnit, the jQuery plugin. Tests are organized by package, e.g., controllers, services, delegates, etc. Each controller, component, service, delegate, and data model has it's own set of unit tests. Event classes are also tested mainly for naming consistency and validation. The test index.jsp reads the same includes.jsp used by the primary application to insure all includes are consistent. The applications MainContainer is also included to insure that components are configured correctly.

Node tests use mocha and should as the test framework. New client projects will also migrate to mocha and chai using grunt as the build tool.

Running Unit Tests

All projects have at least one Makefile that help coordinate building and running of tests. For most projects, you can run "make test" to run unit tests and jshint static file tester. Some projects use grunt to actually run test and jshint, others run tests and jshint without the use of grunt. Those projects require that jshint is installed on the target machine.

Running QUnit Tests

You run unit tests through a browser, preferably Chrome or Safari by accessing the application's top entry point appended with "test". With reference to the html5 project above, web access on the development machine would be through "http://localhost:8080/html5/app" and the test link would be is "http://localhost:8080/html5/test". *Some developers configure their development containers to point to alternate port, such as 3000.*

Continuous Unit Testing

Continuous unit testing is done by appending "?loop=n" to the test url where n is the number of seconds. An example: "http://localhost:8080/html5/test?loop=5". Developers are encouraged to run continuous unit test while developing to get error/success feedback as they save new code.

Command Line Continuous Unit Testing

We are currently searching for a good command line unit testing framework. Candidates include:

- Karma (node based, formerly Testacular)
- Buster.js (node based)
- TestSwarm (John Resig/jquery)
- JsTestDriver (google)

Test framework reference.

Creating/Using Mocks

The most important mock objects are for services. Each service should have it's own mock that implements all it's public methods. The mocked methods should mimic the expected actions for success and failure modes. Two mocks that are used for services are MockAjax and MockXHR. Use of these mocks enables full testing of services without having to actually connect to a remote service.

Integration Testing

At a minimum, all unit tests should pass for each browser. We don't currently support automated integration tests, but some products do have extensive integration test scripts that lead the tester through as many scenarios as possible to insure correct operations.

Source Code Formatting Standards

Standards are influenced by JavaScript Patterns, written by Stoyan Stefanov and coding standards the RP has established for Adobe/Flex projects. All source code should be checked with jslint. The code doesn't have to completely comply with lint, but it often will point out errors that the IDE and/or compiler won't catch.

General Guidelines

- each statement should be terminated by a semi-colon
- code indentation should always be used and should be 4 spaces (not tabs).
- tabs should be converted to spaces
- all code should be encapsulated in a "Class" like function (see examples above)
- each class should be in a separate file
- each service, controller and delegate must have an associated unit test file
- classes names should be capitalized camel case, e.g. RegistrationService, User, ComponentFactory

- class files should match the class name, e.g., RegistrationService should be defined as a function in the file RegistrationService.js
- method names should be camel case, e.g., calculateTotal(), startWorker(), etc
- use the log (see above) and avoid using the native console
- use the triple equals and non-equals operators (=== and !==) to bypass unwanted type coercion
- use {} instead of new Object(). use [] instead of new Array().
- use an array to create long strings rather than multi-line strings with +
- use array iterators when appropriate (filter, forEach, every, map, some, reduce, reduceRight)

Naming Conventions

Use mixed case nouns when naming classes, objects, and variables. Classes begin with uppercase, objects and variables use lower case. Constants use all uppercase and underscores. Use verbs to name message receptors, functions, etc. Accessors that provide method based access to private variables should use get or set with the exception of booleans that may us is or has.

IE Workarounds

We currently include a small library that compensates for IE shortcomings at least for IE7 and IE8. The fixes implement methods in common objects. The short list is:

- String.prototype.trim()
- Array.prototype.indexOf()
- Array.prototype.forEach()
- JSON.parse()
- JSON.stringify()

Boolean Methods

Methods that return boolean values should be named using "is" or "has" or "was", i.e., isValid(), wasReceived(), hasMember(). It is important that these methods return only true or false. A common mistake is to let a boolean method return null or undefined to represent false, but this leads to difficult to find bugs. Consider this code:

```
// BAD CODE!

// the start date/time
var startTime = null;

var hasStarted = function() {
    // if the start time is set, then we have started
    return (startTime && startTime > 0);
}

if (hasStarted) {
    log.info("we have started");
}
```

This example always logs "we have started". Can you see why? The object hasStarted is a function, but the if statement is expecting a boolean true/false. But, there are actually three possible return values: true, false and null. So without the parans (), hasStarted returns the function, which always returns true. With the parans hasStarted() will return null, which is evaluated false, but not definitively.

A much improved implementation is below:

```
// milliseconds, e.g. startTime = new Date().getTime();
var startTime = 0;

var hasStarted = function() {
    return startTime > 0;
}
```

Curly Braces

Braces should be used in all loops and conditional statements. Always place the curly brace on the same line as the statement, never on a line by itself. Curly brace should be proceeded by a single space. Here are a few examples:

```
// private method
var calculateTotal = function(option) {
    if (option) {
        // do something
    } else {
        // do something else
    }

    while (isRunning()) {
        log.debug('running...');
        // do something else...
    }
};
```

Declaring Function Variables

Javascript has no block scope, just function scope, so variables should only be declared inside functions. The lack of block scope also means that variables should be defined at the top of the function, rather than in line. The preferred style defines multiple variables with a single "var" statement, like this:

```
var MyClass = function(options) {
 "use strict";
    var myclass = this,
        log = RemoteLogger.createLogger('MyClass'),
        dispatcher = options.dispatcher;

    // other public access methods
};
```

JSLint/JSHint my complain if you don't use this form.

Creating Very Long Strings

When constructing long strings, you should not use the + operator on multiple lines. It's much more efficient to create an array with a series of "push" calls or in-line ['this', 'that'] then use join to create the string. Here is an example:

```
var myLongString = [
   "the rain in spain",
   "falls mainly on the ground.",
   "the rain in Seattle falls",
   "mainly on the trees."
].join(' ');
```

Commenting

Class Files

At a minimum, each class (psuedo-class) file should have a header that briefly describes the intention of the class. It should also include the date it was created, the author's name, and any dependencies.

Comment Blocks

Normally, blocks of code that are commented out is a bad practice--that's what source code control is for. But, if you do find the need to comment out a block of code, it's better to use the // rather than /* */ form. The reason is that some regular expressions can contain */ and will mess up the compiler.

The /* */ should be used outside of functions with the exception of Class definitions. Inside functions it's better to use multiple single line comments using //.

The // form should always be followed by a single space, e.g. // this is a comment, not //this...
