gularJS - Code Quality Analysis (Linting)

# Motivation

Douglas Crockford explains the motivation for linting JavaScript far better than I can: <http://www.jslint.com/lint.html>. Here is an edited quote from the page:

JavaScript is a sloppy language, but inside it there is an elegant, better language. Linting helps you to program in that better language and to avoid most of the slop ... A lint tool will reject programs that browsers will accept because it defines a professional subset of JavaScript, a stricter language than that defined by [Third Edition of the ECMAScript Programming Language Standard](http://www.ecma-international.org/publications/standards/Ecma-262.htm). The subset is related to recommendations found in [Code Conventions for the JavaScript Programming Language](http://javascript.crockford.com/code.html).

Basically, Linting is just another tool, freely available to us, to help improve code quality.

# Which Tool?

There are a number of JavaScript linting utilities. Here is an overview:

* [JSLint](http://www.jslint.com/): The original JavaScript linting tool was written by Douglas Crockford. While it got the ball rolling, it was not very configurable. So along came Anton Kovalyov...
* [JSHint](http://www.jshint.com/): A more configurable version of JSLint. Here is a link to Anton’s reasons for forking JSLint: <http://anton.kovalyov.net/p/why-jshint/>. While it is configurable, there is no facility in JSHint for creating your own rules.
* [ESLint](https://github.com/nzakas/eslint): Nikolas Zakas wanted to make JSHint even more modular, to plug in his own rules, so he created another project. Here are his [thoughts](https://github.com/nzakas/eslint#frequently-asked-questions). This tool is very cool and uses the [esprima](http://esprima.org/) engine to parse the JavaScript/ECMAScript. You can write your own ELint rules. It looks good but it is, by its own admission, pre-alpha: there are rules missing and the API is not guaranteed to be stable. Also, there are no plugins for the major editors available yet for ESLint.
* [Closure Linter](https://developers.google.com/closure/utilities/): A python utility built by Google to check that JavaScript adheres to the Google styleguide. The dependency on python is a shame, since, apart from the closure compiler, all the other tools we use only have dependency on node.js.
* [Closure Compiler](https://developers.google.com/closure/compiler): An Java utility that compiles JavaScript into better JavaScript. The AngularJS project is using this compiler to minify its built files. The compiler can provide some [limited style checking](https://developers.google.com/closure/compiler/docs/error-ref#warn) but this requires the compiler to run and there is no integration with editors.In th
* e long term, we should look to use ESLint; it aims to achieve rule parity with JSHint by v1.0 but there is currently no published timescale for this release.

ESLint aims to be a superset of JSHint. Migrating JSHint configuration to ESLint configuration will be fairly straightforward. Both have very similar CLI tools and GruntJ S task plugins.

**Based on configurability, minimizing dependencies, stability and integration with code editors, it seems that, right now, it would be best to go with JSHint and monitor ESLint. When ESLint is stable and has the required features we should migrate over.**

# JSHint

JSHint comes packaged with both a command line interface (CLI) that takes the a list of files to be analysed as an argument and a node.js library. Both can be configured via configuration files or diConfiguration

JSHint checks the code against a set of rules. Each of these rules can be turned on and off, arectives in code comments. It also has various plugins for build systems and editors. See the [Plugins](#heading=h.hcnf2ib9q1k7) section below.

nd in some cases configured further. There are two configuration methods:

* **Configuration Files**: JSHint configuration is stored as JSON in a file, which contains an object where, for each property, the name identifies the rule and the value configures that rule. You can specify this file when running JSHint or it can search for a file called .jshintrc. When searching it will look up the folder hierarchy from the current working folder for this .jshintrc file. *(Alternatively, one can specify rules in the package.json file but this is not appropriate for the AnglarJS project because we need different configuration for different parts of the project.)*
* **Code Comments**: when analysing files, the configured rules can be further modified or overridden by “directives”, which are stored as code comments in the code being analysed. This is useful for exceptional situation where you purposefully break one of the analysis rules on a one-off basis.

## Plugins

JSHint has numerous plugins for both build systems and editors/IDEs.

* Build Tool Plugins: AngularJS uses GruntJS to build and there is a JSHint task plugin for GruntJS available: [grunt-contrib-jshint](https://github.com/gruntjs/grunt-contrib-jshint).
* Editor Plugins: Running the analysis inside your editor is the fastest method for discovering code issues as you write or review code. There are a wide range of JSHint plugins for most code editors. See <http://www.jshint.com/install/>

# Implementation

In AngularJS there are different sets of code that should have subtly different configuration, source code, tests, build utilities, helper scripts, generated documentation, etc. We need to be able to analyse all these different code types regularly to ensure their quality.

This section suggests a specific approach to using JSHint for the AngularJS project and what would need to change in the project to accommodate it.

## Running JSHint

Editors should be set up to run JSHint on any file that is being edited. Developers should ensure that there are no JSHint warnings when writing or reviewing a code file.

When building the project there should be build tasks that run JSHint to analyse every javascript file in the project. We should analyse each of the files in the `src` and `test` folders but also the utility files used to do the build and the helper scripts.

## Configuration

We should configure the rules for JSHint through .jshintrc files placed at suitable points in the folder structure. Specific rules should be overridden on a case-by-case basis through JSHint code comment directives. This allows the following:

* Editor plugins to find the appropriate configuration for the file that is being edited.
* The grunt build task to run different configurations for code that requires similar rules.

## Project Folder Structure

Since the JSHint will search up the folder structure to find a .jshintrc configuration file, it makes sense to structure the source-code folders so that code with common rule configuration (shared globals, for instance) are grouped in the same folder.

### Source Files

Right now the “external” modules source code (ngRoute, ngResource, etc) is placed in the same folder as the core source code (`src/`). To specify configuration for this via .jshintrc files for this folder structure causes unwanted duplication. We would need the following .jshintrc files:

* src *(core .jshintrc)*
  + ng
  + ngAnimate *(external module .jshintrc)*
  + ngCookies *(external module .jshintrc)*
  + ngLocale *(external module .jshintrc)*
  + ngMock *(external module .jshintrc)*
  + ngResource *(external module .jshintrc)*
  + ngRoute *(external module .jshintrc)*
  + ngSanitize *(external module .jshintrc)*
  + ngScenario *(ngScenario .jshintrc)*
  + ngTouch *(external module .jshintrc)*
  + *various core files such as angular.js, ...*

Note that the *external module .jshintrc* must be repeated for each module. We can simplify this if we move the folders and files from the root of the `src` folder into subfolders:

* src (basic .jshintrc)
  + core *(core .jshintrc)*
    - ng
    - AUTO
    - *various core files such as angular.js, ...*
  + modules *(external module .jshintrc)*
    - ngAnimate
    - ngCookies
    - ngLocale
    - ngMock
    - ngResource
    - ngRoute
    - ngSanitize
    - ngTouch
  + ngScenario (*scenario .jshintrc*)

This folder structure groups files by their type so that they can share rules:

* The modules folder holds all the external modules. We just have a single configuration file for all the modules inside.
* Files such as angular.js and jqLite, etc, are placed within core folder with the core .jshintrc configuration file.
* The ngScenario code has its own folder - it is not actually an Angular module and it needs a different configuration file to the modules.

### Test Files

A similar folder structure to the src folder, above, should be provided for the code in the test folder.

### Generated Documentation Files

The AngularJS documentation contains lots of example code. This code is held inside the “ngdoc” documentation themselves. This area is traditionally an area where code errors easily slip in. Currently, we run any e2e tests found in these examples, but no unit tests.

The AngularJS documentation website is an AngularJS application itself, which exposes and runs the example code on the fly. Since the example code is stored inside the HTML files JSHint cannot directly parse and analyse this code.

In the future, we need a way to write and test code examples separately from the code itself. Maybe we could create a build task/script to extract example code from documentation files, then analyse and test them independently. Or maybe example code should be moved into their own files - but this then leads to additional complexity for linking them up and keeping them in synch.

### Library Files

There is a lib folder containing local libraries that have not been imported using npm or bower. These should be analysed.

### Other Files (Build, Scripts, etc)

The following folders contain code that could be analysed:

* docs
  + components
  + component-spec
  + src
  + spec
* example
* i18n
  + src
  + spec
  + e2e

These files should probably be moved out into separate projects:

* We should consider moving the documentation generation code and the i18n import code out into their own projects and importing them into the AngularJS project as npm modules, similar to Karma and what will happen with ngScenario -> Protractor.
* Are the files in the example folder used anywhere? The buzz example is clearly out of date (and has been for over a year). Perhaps this should be removed from the project altogether?
* Is lib/htmlparser/htmlparser.js used anywhere? The code is duplicated inside ngSanitize/sanitize.js.

There are also various utility scripts in the root folder as well, such as changelog.js, compare-master-to-stable.js, etc.

## Grunt Tasks

The [grunt-contrib-jshint](https://github.com/gruntjs/grunt-contrib-jshint) plugin provides a [GruntJS multi-task](https://github.com/gruntjs/grunt/wiki/Configuring-tasks#task-configuration-and-targets) to run JSHint against an arbitrary set of targets. Each target will take a list of files to analyse and a set of options. For example:

{

jshint: {

src: {

core: {

files: files['angularSrc'],

jshintrc: ’src/core/.jshintrc’

},

modules: {

files: files[angularSrcModules'],

jshintrc: ’src/modules/.jshintrc’

},

ngScenario: {

files: files['angularScenario'],

jshintrc: ’src/ngScenario/.jshintrc’

},

...

},

test: {

...

}

}

}

### Task Targets

The angularFiles.js file should be used to generate the list of files for each target. Currently, angularFiles is coded primarily to support creating the karma configuration files. This should be refactored to expose the lists of the test files separately from the files only needed for the karma tests.

### Task Options

The following options should be provided in the grunt confguration:

* **jshintrc**: Specify what .jshintrc file to load for this target
* **reporter**: what reporter to use to format the result from the analysis
* **force**: whether to fail the task if the code breaks the rules.

Initially, the force option should be set to true. This is similar to how minErr reporting works right now. The task would give feedback during builds on what is breaking the JSHint rules, while not preventing the build from completing.

Once the project code base has been cleaned up and is not breaking the rules, the force option can be turned off to ensure future commits do not introduce invalid code.

# What Rules To Enforce?

The list of JSHint options/rules can be found here: <http://www.jshint.com/docs/options/>. There is not a one-to-one correspondence between the AngularJS coding standards and the JSHint rules. Until ESLint is ready we must pick the JSHint rules to enforce that provide the maximum improvement in code quality without breaking what the AngularJS project considers valid code.

Rules should be turned on one at a time and the code that breaks the rule should be fixed or the rule should be disabled for that particular instance. Here is a list of a basic rules, which should be applied across the board for jshint:

* **bitwise**: prohibits the use of bitwise operators such as ^ (XOR), | (OR) and others. Bitwise operators are very rare in JavaScript programs and quite often & is simply a mistyped &&. The AngularJS src folder only contains three valid instances of a bitwise operator, which could be overridden inline.
* **immed**: prohibits the use of immediate function invocations without wrapping them in parentheses.
* **newcap**: requires you to capitalize names of constructor functions.
* **noarg**: prohibits the use of arguments.caller and arguments.callee.
* **noempty**: warns when you have an empty block in your code.
* **nonew**: prohibits the use of constructor functions for side-effects.
* **trailing**: makes it an error to leave a trailing whitespace in your code.
* **maxlen**: lets you set the maximum length of a line.
* **boss**: suppresses warnings about the use of assignments in cases where comparisons are expected. This is done often in the AngularJS code base.
* **eqnull**: suppresses warnings about == null comparisons. This could eventually be removed.
* **expr**: suppresses warnings about the use of expressions where normally you would expect to see assignments or function calls. We often have code that looks like this:  
   event.target && event.target.blur();  
  where we are calling a method with a side-effect but only if the object containing the method exists.
* **globalstrict**: suppresses warnings about the use of global strict mode.
* **laxbreak**: suppresses most of the warnings about possibly unsafe line breakings in your code.
* **loopfunc**: suppresses warnings about functions inside of loops.
* **sub**:suppresses warnings about using [] notation when it can be expressed in dot notation. This happens a lot in the AngularJS codebase.

The following rules could be introduced but would have a large impact on the code base:

* **camelCase**: allows you to force all variable names to use either camelCase style or UPPER\_CASE with underscores.
* **eqeqeq**: prohibits the use of == and != in favor of === and !==.
* **forin**: requires all for in loops to filter object's items.
* **indent**: enforces specific tab width of 4 for your code.
* **latedef**: prohibits the use of a variable before it was defined.
* **quotemark**: enforces the consistency of quotation marks used throughout your code.
* **undef**: prohibits the use of explicitly undeclared variables. Variables defined outside the current file must be declared using the **globals** jshint directive or configuration.
* **unused**: warns when you define and never use your variables. Variables that will be used outside the current file must be declared using the **export** jshint directive.
* **maxcomplexity**: lets you control cyclomatic complexity throughout your code. This would be interesting…

The following rules are not compatible with the AngularJS code style:

* **curly**: requires you to always put curly braces around blocks in loops and conditionals.
* **es3**: tells JSHint that your code needs to adhere to ECMAScript 3 specification.
* **plusplus**: prohibits the use of unary increment and decrement operators.
* **strict**: requires all functions to run in ECMAScript 5's strict mode. We will use **globalstrict** instead.
* **maxparams**: lets you set the max number of formal parameters allowed per function.
* **maxdepth**: lets you control how nested do you want your blocks to be.
* **maxstatements**: lets you set the max number of statements allowed per function.
* **asi**: suppresses warnings about missing semicolons.
* **debugger**: suppresses warnings about the debugger statements in your code.
* **esnext**: tells JSHint that your code uses ECMAScript 6 specific syntax.
* **evil**: suppresses warnings about the use of eval. There are very few places in the code base that use eval and those are important and should be identified with specific JSHint overrides.
* **funcscope**: suppresses warnings about declaring variables inside of control structures while accessing them later from the outside.
* **iterator**: suppresses warnings about the \_\_iterator\_\_ property.
* **lastsemic**: suppresses warnings about missing semicolons, but only when the semicolon is omitted for the last statement in a one-line block.
* **laxcomma**: suppresses warnings about comma-first coding style.
* **moz**: tells JSHint that your code uses Mozilla JavaScript extensions.
* **multistr**: suppresses warnings about multi-line strings.
* **proto**: suppresses warnings about the \_\_proto\_\_ property.
* **scripturl**: suppresses warnings about the use of script-targeted URLs—such as javascript:....
* **smarttabs**: suppresses warnings about mixed tabs and spaces when the latter are used for alignmnent only.
* **shadow**: suppresses warnings about variable shadowing i.e. declaring a variable that had been already declared somewhere in the outer scope.
* **supernew**: suppresses warnings about "weird" constructions like new function () { ... }and new Object;.
* **validthis**: suppresses warnings about possible strict violations when the code is running in strict mode and you use this in a non-constructor function.

## Declaring Globals

A major element of configuring JSHint is to define what variables should appear on the global scope. We can specify these globals inside the appropriate .jshintrc files. When we specify globals the name of the property is the name of the global, the value is either true or false. True means that the value is a global object. False means it is global **and** read-only.

### Standard Globals

JSHint comes with various sets of globals configurations tailored for specific environments. These can be turned on with special JSHint directives:

* browser: variables commonly provided by a web browser environment.
* node: variables provided by the node.js environment.

### Custom Globals

The final distributable AngularJS files only expose a single variable to the global scope: **window.angular**. Since the code base is actually built from lots of small files that are concatenated together, each file appears to expect numerous methods and properties to be on the global scope. This is especially true of the core files.

All the tests in AngularJS use the Jasmine library and ngMocks, which themselves provide a number of global variables.

* **core source files**: use a large number of global variables, such as toBoolean, hashKey, forEach, etc. Also all the [standard browser globals](https://github.com/jshint/jshint/blob/master/src/vars.js#L47).
* **angular test files**: these files need to have the same globals available as their corresponding source file. In addition, they use Jasmine and ngMock globals, such as describe, it, beforeEach, jasmine, createSpy, inject and module.
* **module source files**: should only have angular and the [standard browser globals](https://github.com/jshint/jshint/blob/master/src/vars.js#L47).
* **utility source files**: use [node.js globals](https://github.com/jshint/jshint/blob/master/src/vars.js#L374), such as require, module, setTimeout, etc.
* **utility test files**: use Jasmine and node.js globals.

Standard globals such as those used in the browser or node.js are provided by JSHint. We can simply set the appropriate directive in the .jshintrc:

# Rollout Strategy

There are many places in the AngularJS codebase that would break the suggested rules. It would be a very large effort to implement JSHint throughout the codebase and in the build in one go. This section lays out a strategy for implementation of JSHint.

## Trial JSHint on a module

The module to trial should be large enough to provide realistic feedback but not so huge or complex that it is too much effort to implement quickly.

* Install the JSHint CLI tool:  
  npm install -g jshint
* Install JSHint plugins for your favourite editor
* Create a branch:  
  git checkout -b lint master
* Create a local .jshintrc for the module’s folder  
  Add necessary globals for this module
* Update gruntFile to run the JSHint grunt task  
  Set it to check only the chosen module
* Use your editor and the jshint CLI tool to identify issues.
* Run grunt jshint to see how it pans out.

## Implement Source Code Analysis

Set up the rest of the source code to be analysed by a grunt task

* Reorganize the source and test folders to accommodate .jshintrc files.
* Add minimal .jshintrc files and configuration to grunt to analyse all the folders under src
* Run the JSHint in **forced** mode so that it doesn’t break the build, but just reports the issues

### Implement Rules Step by Step

* Turn on rules, fixing bad code (or overriding warnings for good code) as it is identified.
* Merge “style” commits to master as necessary

## Implement Test Code Analysis

## Implement Utlity Code Analysis