My minimal toolkit for development

The argument about tool overload is not a new one. There have been many discussions over the years about what framework is better, what language is better, whether you should use Ruby on Rails versus Angular, SASS instead of LESS or Compass instead of plain SASS and even whether you should use SASS syntax over SCSS or vice versa.

I've been working with web technologies since before we had Javascript and CSS. I subscribe to the school that says learn the basics and then jumpt into the fancy advanced stuff or in this case, paraphrasing Addy Osmani: `first do it, then do it right, then do it better` and then move on to the next thing.

For me that means: Learn how to use HTML, CSS and vanilla Javascript before jumping into whatever framework or library is popular at the moment.

Basic Infrastructure and build system installation and configuration

To start we'll define these core tools:

- Gulp as my task runner
- ESLint as my Javascript Code Quality checker
- Babel as my transpiler
- Accessibility evaluation
- Ruby SASS as my SASS compiler / interpreter

Rather than have all the tools available right away we'll build the infrastructure based on what we need to do.

In addition we have a few optional libraries and dependencies.

- Polymer as my development framework
- Typescript as a second source of scripts and functionality

We'll discuss the optional elements later.

Before we start we need to initialize the node `package.json` file.

```
npm init
```

> Note that the tasks described below are just the tasks. If you want to see the full Gulpfile.js, see the full Gulpfile in the associated Github repository.

Gulp

When I started looking at modern development Grunt was the favorite (and perhaps the only) task runner available. I moved to Gulp because I found it easier to work with (you can create custom task easier than you can in Grunt) and because Polymer was using Gulp on their build system.

After complaining for a while I realized the advantages of Gulp both in terms of configuration and in terms of extensibility (something I still don't do much of). Tasks in Grunt are very procedural and, as a result,

```
npm install -g gulp
npm install --save-dev gulp
```

ESLint

When I first started I used a combination of JSHint and JSCS to keep my code clean and the syntax working. When JSCS announced their merging with ESLint I decided it was time to change.

ESlint has presets available. I've chosen to use Google's preset, both because it was the requirement when working on Progressive Web Applications at Google and because it matches a lot of what I do when writng Javascript.

```
npm install -g eslint
npm install --save-dev eslint
```

```
npm install --save-dev eslint-config-google
```

Babel and Presets

Babel (formerly known as 6to5) transpiles ES2015 and later into ES5 that can be used in modern browsers. In addition to installing the Command Line Tool we need to install presets to add specific features that we want to work with.

Each preset contains the features that made it to stage 3 for that given version of EcmaScript. This will allow developers to work with the features of each version of the ECMAScript standard before they are fully implemented in browsers.

Some people may argue that we don't need the `es2015` preset because all modern browsers are above 95% support according to Kangax's ES2015 compatibility table. That is true only doe

For preset-es2015 the supported plugins are:

- check-es2015-constants
- transform-es2015-arrow-functions
- transform-es2015-block-scoped-functions
- transform-es2015-block-scoping
- transform-es2015-classes
- transform-es2015-computed-properties
- transform-es2015-destructuring
- transform-es2015-duplicate-keys
- transform-es2015-for-of
- transform-es2015-function-name
- transform-es2015-literals
- transform-es2015-modules-commonjs
- transform-es2015-object-super
- transform-es2015-parameters
- transform-es2015-shorthand-properties
- transform-es2015-spread
- transform-es2015-sticky-regex
- transform-es2015-template-literals
- transform-es2015-typeof-symbol
- transform-es2015-unicode-regex
- transform-regenerator

For preset-2016 the supported plugin is:

- transform-exponentiation-operator

For preset-2017 the currently supported plugins are:

- syntax-trailing-function-commas
- transform-async-to-generator

Note that there may be additional transformations that will still make it to the ES2017 specification.

```
npm install -save-dev babel-cli
npm install --save-dev babel-preset-es2015
npm install --save-dev babel-preset-es2016
npm install --save-dev babel-preset-es2017
```

Accessibility

Skipping or not paying attention to accessibility is easy. I got a harsh lesson when I was working on a Polymer project and got absolutely no keyboard navigation. To make it harder to ignore I've created a task using a11y to evaluate accessibility as part of the build process.

The pluin may change (it hasn't been updated in a while) but the underlying tool will not.

```
npm install --save-dev gulp-a11y
```

Installing additional Gulp plugins

There are plugins we'll use throughout multiple tasks so rather than install them multiple times we'll install them once here and not have to worry about it again.

`gulp-sourcemaps` generates sourcemaps for both CSS Stylesheets and JS scripts. This allows Dev Tools on your browser to translate locations in a minified file to the original script, making it easier to debug and troubleshoot problems in your code.

`gulp-sizes` will give you the size of the completed task.

`gulp-sequence` allows gulp to run tasks synchronously as opposed to de default to run as many task as possible concurrently.

`del` provides a delete function that can be incorporated into other tasks. We'll use it to create a clean task to reset our working directory.

`gulp-load-plugins` loads all packages defined in `package.json` and assigns them to an object of your choice, usually `$`. The limitation is that the package must start with the string `gulp-` for load-plugins to pick it up.

```
npm install --save-dev gulp-sourcemaps
npm install --save-dev gulp-sizes
npm install --save-dev gulp-sequence
npm install --save-dev del
npm install --save-dev gulp-load-plugins
```

editorconfig

It also doesn't hurt to have an editorconfig configuration file in your project. This will take care of indenting and line endings for you. It can be configured to treat different languages differently and it even works with specific files or globs for directories.

In the example below (taken from the editorconfig site) we work with Javascript and Python files with different indentations and charsets. It will also take care of converting all the line endings to Unix and adding an extra line at the end of the file as required by the Google ESLint rules.

```
# EditorConfig is awesome: http://EditorConfig.org

# top-most EditorConfig file
root = true

# Unix-style newlines with a newline ending every file
[*]
end_of_line = lf
```

```
insert_final_newline = true

# Matches multiple files with brace expansion notation
# Set default charset for Javascript and Python
[*.{js,py}]
charset = utf-8

# 4 space indentation for Python files
[*.py]
indent_style = space
indent_size = 4

# Indentation override for all JS
[**/*.js]
indent_style = space
indent_size = 2
```

Configuring the basics

Now that we've installed the basic tools we need to create configuration files for them. These configurations will make use of the tools we instaled in the previous step and make sure we stay clean and we can concetrate on the work we want to do.

Configuring Babel

`.babelrc` is the configuration file for Babel. Ours is as simple as it can be: we use the presets for es2015, 2016 and 2017. We'll repeat this configuration in the Babel task but it's still good to have it since there may be situations when we run Babel from the command line.

```
{
  "presets": [
    "es2015",
   "es2015"",
    "es2017"
  ]
```

```
}
```

ESLint configuration

`.eslintrc.json` is the primary configuration file for ESLint. In it we have 3 areas:

- **extends** tells eslints which plugins to extend from. In this case the base recommended plugin as well as Google's eslint extensions and rules
- **env** indicates what environments to accept. In this example we use es6, browser, Node, jQuery and Mocha
- **rules** tells ESLint what rules we want to override. Some of these overrides work in conjunction with the rules we set up in our `.editorconfig` configuration file

```
{
    "extends": [
        "recommended",
"recommended"le"
    ],
    "env": {
        "es6": true,
        "browser": true,
        "node": true,
        "jquery": true,
        "mocha": true
    },
    "rules": {
        "indent": ["error", 2],
        "error""linebreak-style"or", "unix"],
        "quotes", ""quotes" "single"],
        "no-cond-as"single""no-cond-assign""],
        "no-console": "off"": ""no-console": "off"
    }
}
```

Layering on top of the basics

Now that I have all the tasks set up and configured we can start building the tasks for Babel and ESLint. Because we've installed all the necessary plugins we'll just look at the tasks that will run as part of our `gulpfile.js`.

Babel Gulp Task

The Babel task will perform the following steps

- Take all the files under `src/js` that end with `.js` to any directory depth
- Initialize the sourcemaps engine
- Run Babel using the indicated presets
- Write the sourcemap
- Write the ES5 transpiled code to the destination directory (`dest/js`)

```
gulp.task('babel', function() {
  return gulp.src('src/js/**/*.js')'src/js/**/*.js'rcemaps.init())
    .pipe($.babel({
      presets: ['es2015', 'es2016', 'es2017']
    }))
    .pipe($.sourcemaps.write('.'))
    .pipe(gulp.dest('dest/js/'))
    .pipe($.size({
      pretty: true,
      title: 'Babel'
    }));
});
```

ESLinting your scripts

ESLint is a combination of Linter and Quality control tool. It took the place of both JSHint and JSCS once the later merged into ESLint. The task below runs ESLint on all the Javascript files on your application except for the `node_modules` tree. For each file it logs information to console about warnings, errors and how many messages it generated.

```
gulp.task('eslint', function () {
  gulp.src(['**/*.js','!node_m'**/*.js'/**'])
    .pipe(eslint({
       configFile: '.eslintrc.json'
    }))
    .pipe(eslint.result(function(result) {
        console.log('ESLint result: ' + result.filePath);
        console.log('# Messages: ' + result.messages.length);
        console.log('# Warnings: ' + result.warningCount);
        console.log('# Errors: ' + result.errorCount);
    }));
});
```

Accessibility

To keep myself honest when it comes to accessibility I created

```
gulp.task('audit', function () {
  return gulp.src('./**/*.html')
  './**/*.html'))
    .pipe(a11y.reporter());
});
```

Purpose tasks

Most of Gulp's (and other task runners) workflow is to perform repetitive of time consuming tasks or tasks that need to be continuously repeated during development. The tasks below represent some of these activities.

Each of the tasks include a list of Gulp plugins to install and the actual task we include in the `gulpfile.js` that we'll run to complete the tasks.

> Note that the tasks described below are just the tasks. If you want to see the full Gulpfile.js, see the full Gulpfile in the associated Github repository.

Development Server and Content Preview

For our development server we'll use Browsersync. It allows multiple synchronized tests from the same content (we can open Safari, Chrome and Firefox and whenever files change, all the browsers will be notified).

The `connect-history-api-fallback` deals with the fact that Single Page Applications (SPA) typically only utilise one index file that is accessible by web browsers: usually index.html. Navigation in the application is then commonly handled using JavaScript with the help of the HTML5 History API. This results in issues when the user hits the refresh button or is directly accessing a page other than the landing page, e.g. /help or /help/online as the web server bypasses the index file to locate the file at this location. As your application is a SPA, the web server will fail trying to retrieve the file and return a 404 - Not Found message to the user.

While not all the projects I work in are SPAs it's good to have it.

```
npm install --save-dev browser-sync
npm install --save-dev connect-history-api-fallback
```

The task performs the following actions:

- Configures Browsersync
- Adds the snippet string to match and the function to insert the snippet into the page
- Has a commented out option to use HTTPS
- Configures the base directories for the server
- Specifies the middleware to use (historyApiFallback)
- Set up watches to reload the page when something changes.
    - If the SASS changes we run the sass compilation and the CSS processing tasks before reloading

```
var reload = browserSync.reload;

gulp.task('serve', function() {
  browserSync({
    port: 2509,
```

```
    notify: false,
    logPrefix: 'ATHENA',
    snippetOptions: {
      rule: {
     'ATHENA': '<span id="browser-sync-binding"></span>',
        fn: function (snippet) {
          return snippet;
        }
      }
    },
    '<span id="browser-sync-binding"></span>'// Run as an https by uncomme
    // will present a certificate warning in the browser.
    // https: true,
    server: {
      baseDir: ['.tmp', 'dest'],
      middleware: [historyApiFallback()]
    }
  });

  gulp.watch(['app/**/*.html'], reload);
  gulp.watch(['app'.tmp'// This uses an unsigned certificate which on firs
  gulp.watch(['app/css/**/*.scss'], ['sass', 'processCSS', reload]);
  gulp.watch(['app/images/**/*'], reload);
});
```

SASS Conversion and CSS Processing

I've chosen to use the SCSS version of SASS as my primary stylesheet language because of its flexibility and the programming-like structures available. It deviates from the KISS principle but the features make it a worthy tradeoff.

SASS was originally developed as a Ruby library and is available as a Ruby Gem. This introduces an additional dependency but the Node version (libsass) still chokes on older projects so I'm ok with using the Ruby version.

The SCSSLint library also depends on Ruby

To install the Ruby dependencies, run the following commands

```
gem install sass
gem install scss_lint
```

Then install the node packages as usual:

```
npm install --save-dev gulp-ruby-sass
# If you don't want to depend on Ruby run the following command instead
npm install --save-dev gulp-sass
# If you don't want to depend on Ruby don't run the next two lines
npm install --save-dev scss-llint
npm install --save gulp-scss-lint-stylish2
npm install --save-dev autoprefixer
npm install --save-dev gulp-postcss
```

The `sass` task takes all the files in the path startinin the `scss` directory and do the following:

- Run SCSS lint to make sure there are no errors
- Run the SASS compiker usin the expanded syntax

```
gulp.task('sass', function() {
  return sass('app/scss/**/*.s'app/scss/**/*.scss'emap: true,
      style: 'expanded'})
    .pipe($.scsslint({
      'reporterOutputFormat': 'Checkstyle'
     }))
    .pipe(gulp.dest('app/css/expanded'))
    .pipe($.size({
      pretty: true,
      title: 'SASS'
    }));
});
```

We are ready to process the resulting CSS.

We first set a variable with an array of the browser versions we want to auto

prefix for. This was originally taken from the Polymer Starter Kit gulpfile.

We then run the following tasks:

- Autoprefix the CSS using the browser versions defined in the `AUTOPREFIXER_BROWSERS` variable
- CleanCSS to minimize the result

```
var AUTOPREFIXER_BROWSERS = [
  'ie >= 10',
  'ie_mob >= 10',
  'ff >= 30',
  'chrome >='ie_mob >= 10'i >= 7',
  'opera >= 23',
  'ios >= 7',
  'android >= 4.4',
  'bb >= 10'
];

gulp.task('processCSS', function() {'opera >= 23'p.src('src/css/**/*.css'
    .pipe($.sourcemaps.init())
    .pipe($.postcss([
      autoprefixer({ browsers: AUTOPREFIXER_BROWSERS })
    ]))
    .pipe(cleanCSS({
      advanced: false,
      aggressiveMerging: false,
      debug: tr'src/css/**/*.css'ks: true,
      mediaMerging: false,
      processImport: false,
      rebase: false
    }))
    .pipe($.sourcemaps.write('.'))
    .pipe(gulp.dest('dist/css'))
    .pipe($.size({
      pretty: true,
      title: 'processCSS'
    }));
  });
```

Creating images for responsive images elements

Responsive Images deal with the fact that Sixty-two percent of the weight of the web is images, and we're serving more image bytes every day regardless of the device accessing our web content. This is particularly important in emerging markets where bandwidth is expensive and one 2MB page can take 20% of a user's monthly bandwidth or more.

There are different ways to create responsive images documented in many places. A List Apart's Responsive Images in Practice and its update do a good job of explaining

The task creates a set of reponsive images for each of the PNG and JPG images in the images directory to any depth, except for the images in the touch directory (we only need one version of each image in that directory). We currently only create jpg images. We could take responsive images to the extreme and also generate webp versions for browsers that support the format.

This task does not create the markup we need to work with responsive images. For a more detailed look at responsive images and the necessary markup, see Learning abut responsive images

```
gulp.task('processImages', function() {
  return gulp.src(
    [
      'app/images/**/*.{jpg, pn'app/images/**/*.{jpg, png}'h/*.png'
    ])
    .pipe($.responsive({
      '*': [{
        // image-small.jpg is 200 pixels wide
        width: 200,
        rename: {
          suffix: '-small',
          extname: '.jpg'
        }
      }, {
        // image-small@2x.jpg is 400 pixels wide
```

```
      width: 200 * 2,
      rename: {
        suffix: '-small@2x',
        extname: '.jpg'
      }
    }, {
      // image-large.jpg is 480 pixels wide
      width: 480,
      rename: {
        suffix: '-large',
        extname: '.jpg'
      }
    }, {
      // image-large@2x.jpg is 960 pixels wide
      width: 480 * 2,
      rename: {
        suffix: '-large@2x',
        extname: '.jpg'
      }
    }, {
      // image-extralarge.jpg is 1280 pixels wide
      width: 1280,
      rename: {
        suffix: '-extralarge',
        extname: '.jpg'
      }
    }, {
      // image-extralarge@2x.jpg is 2560 pixels wide
      width: 1280 * 2,
      rename: {
        suffix: '-extralarge@2x',
        extname: '.jpg'
      }
    }, {
      // Global configuration for all images
      // The output quality for JPEG, WebP
      // and TIFF output formats
      quality: 80,
```

```
        // Use progressive (interlace) scan
        // for JPEG and PNG output
        progressive: true,
        // Skip enalrgement warnings
        skipOnEnlargement: false,
        // Strip all metadata
        withMetadata: true
      }]
    })
      .pipe(gulp.dest('dist/images')));
})
```

Image compression

There are times when using responsive images is overkill. Some times it's enough to just reduce the size of an image withough loosing quality. Imagemin will optimize SVG, JPG, PNG and GIF and, where appropriate, convert the image to WebP.

```
npm install --save-dev gulp-imagemin
npm install --save-dev gulp-svgmin
npm install --save-dev gulp-svgstore
npm install --save-dev imagemin-mozjpeg
npm install --save-dev imagemin-webp
```

The task will take all the images in the `src/images/` directory tree and first pass them through imagemin using the mozjpeg compressor and then convert them to WebP.

```
gulp.task('imagemin', function() {
  return gulp.src('src/images/**')
   'src/images/**'({
      progressive: true,
      svgoPlugins: [
        {removeViewBox: false},
        {cleanupIDs: false}
```

```
      ],
      use: [mozjpeg()]
    }))
    .pipe(webp({quality: 50})())
    .pipe(gulp.dest('dist/images'))
    .pipe($.size({
      pretty: true,
      title: 'imagemin'
    }));
});
```

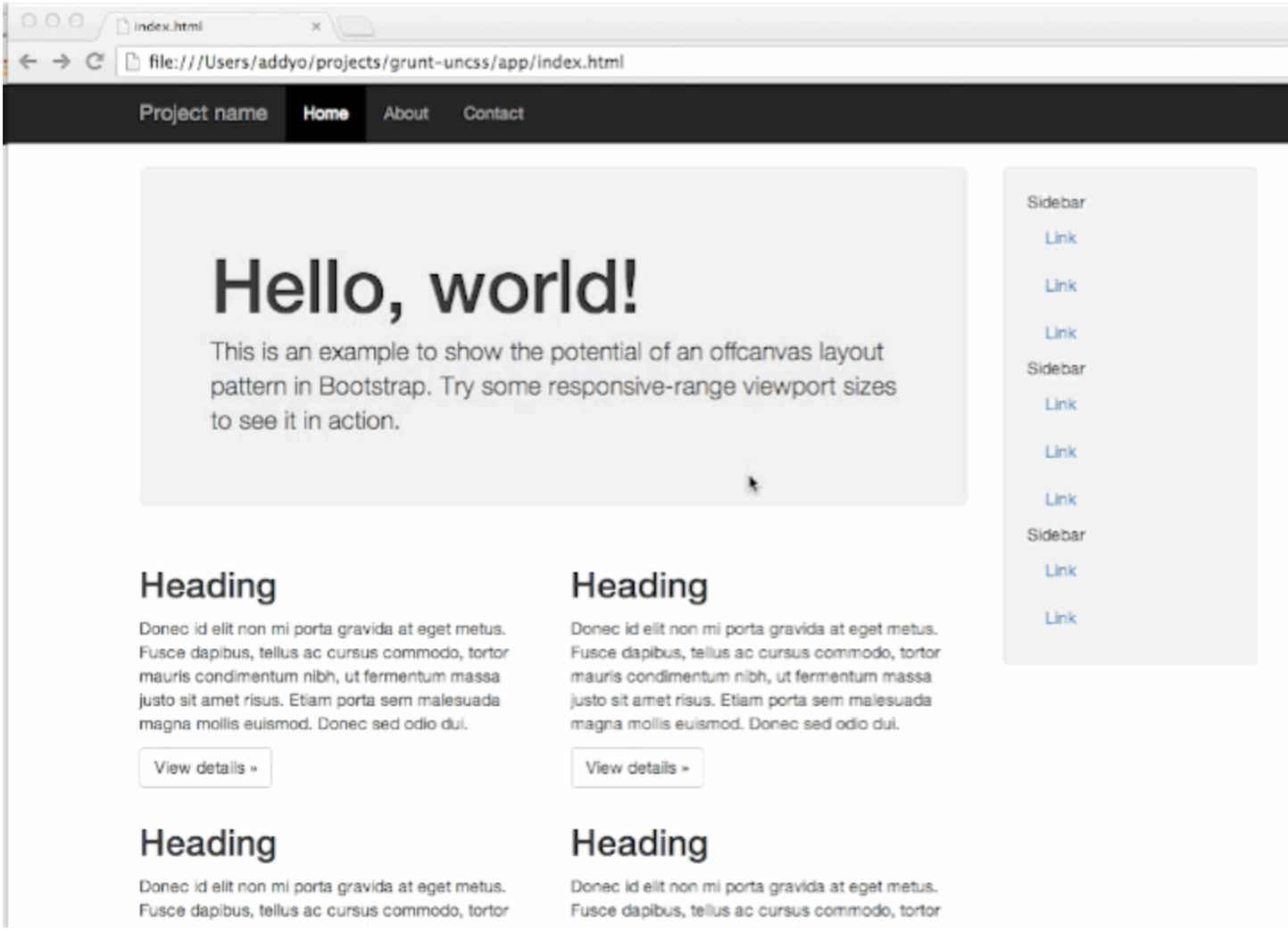
UNCSS

When using the same CSS for multiple project it’s impossible not to bloat the libraries we use.

This is particularly true when using libraries like Bootstrap or Foundation with their big multipurpose scripts which we seldom slim down, even when building custom versions of the libraries.

UNCSS will take one ore more CSS files and match it with all the content of one or more HTML files and produce a new CSS style sheet with only the selectors used in the HTML files. This will definitely reduce the size of the CSS files and make the site/app load faster.

In the example below, the bootstrap CSS was reduced to 11kb, a 10x reduction in its weight.




```
gulp.task('uncss', function() {
  return gulp.src('app/css/**/*.css'app/css/**/*.css'cat('main.css'))
    .pipe($.uncss({
      html: ['index.html']
    }))
    .pipe(gulp.dest('dist/css/main.css'))
    .pipe($.size({
      pretty: true,
      title: 'Uncss'
    }));
});
```