Gulp

NPM Packages

Gulp Tutorials (multi-part)

MahsupJS’s Gulp

TODO:

* **~~Update the Sequence and Parallel Task Processing section~~**
* **~~Pre-populating the Angular template cache~~**
* **Create a project to work with in the tutorial. Make a Yomen install & instruct how to pull from GitHub.**
  + **Possibly create a repository just for this tutorial.**
* **~~Remove un-used css~~**
* **Go through each part and update with the new code.**
* I’m not sure what this is about but it seems important.
* Add static analysis for cyclomatic complexity and stuff.
* Read through document and add consistency
* Go through and remove “./”. Start with the directory name instead. Works ok in tasks but not in watch so change all to be consistent.
* Add code consistency with './dist/'
* Go through code one last time to make sure the doc is updated.
* Walk through Tutorial from beginning to end to make sure it works.
* Squeeze out a little more performance. See what I can concatenate safely.
* Add explanation of --save-dev
* Add explanation on why we install gulp globally and locally
* Add comment on being in the root when executing npm install commands.
* Add comment on easily getting a process window in the root.
* Add comment about the difficulty of removing node\_modules folder
  + Future versions will be flatter and more windows friendly.
* Add the CSS remove task.
* Watch this before publishing this blog(s) <http://www.pluralsight.com/courses/javascript-build-automation-gulpjs>

# Gulp Tutorials

## GULP Tutorial Part 1 – Reasons for Build Tools like Gulp

### Productivity

**The reason for a build system is always productivity. Otherwise we wouldn’t invest time into it.**

**Build systems perform house cleaning work allowing you to focus on code. Before build systems, if you were lucky, you could right click and select “minify” in your IDE. As lucky as this might have been minification might not have been worth the additional development effort required. Build systems solve this.**

**Build systems perform tasks with a level of precision humans are incapable of. In order for Continuous Integration and Continuous Delivery to work a build system must be used to keep the human element out. Continuous Delivery requires automation at all levels including testing to mitigate common deployment defects.**

**There are thousands of plugins to perform just about any task imaginable. Here are a few.**

**Performance/Optimization**

* **Minification of JavaScript files**
* **Minification of CSS files**
* **Slimming down CSS classes that are not used**
* **Concatenating many JavaScript files to reduce get requests**
* **MAP files created for debugging at run-time**

**Deployment**

* **Files can be optimized then copied to a folder to isolate deployment from development**
* **A zip file can be generated for deployment**
* **Automated tests can be executed**
* **Create deployments with a particular purpose; IE: an app can be built for mobile**

**Static Analysis**

* **Linters can be executed against your code producing advice**
* **Cyclomatic complexity and other measures can be generated.**

**Documentation**

* **Documentation can be generated from code into readable formats.**
* **HTML documents can be generated from Markdown, a popular text format.**

**Additional resources**

<https://www.youtube.com/watch?v=XJ5F-Auhato>

## GULP Tutorial Part 2 – Gulp Setup

Download and install NodeJS or IoJS. IoJS is a fork of NodeJS by many in the community who wanted more responsive Node advancement. Theoretically you should be able to use either for the Mashup.

<https://nodejs.org/>

<https://iojs.org/en/index.html>

npm packages are defined in the package.json file.

To create a package.json file

npm init

**Installing NPM Packages**

At this point Gulp can be installed with the following command. Notice the “-g” command. This causes the npm package to be deployed globally.

Installing Gulp (add “-g” to install globally)

npm install -g gulp

For more information on installing npm packages  
<https://docs.npmjs.com/getting-started/installing-npm-packages-locally>

Create a basic gulpfile.json with the following content, in the root of your project. Add the following scaffolding to the new gulpfile.json file.

var gulp = require('gulp');

gulp.task('default', function() {

// place code for your default task here

});

## GULP Tutorial Part 3 – Adding Plugins

Plugins provide function to the task runner.

TODO: How to search for plugins. Consider the number of downloads and activity on github when trying to determine which plugin to choose.

You can search for plugins here  
<http://gulpjs.com/plugins>

Once you’ve found a plugin navigate to the plugins page. Here you’ll find general information on how to use the plugin and usually a couple examples to get you started.



Install a few useful plugins from the commandline of the root of your project.

npm install gulp --save-dev

npm install gulp-uglify --save-dev

npm install gulp-rename --save-dev

npm install gulp-rename --save-dev

npm install gulp-sourcemaps --save-dev

To add these plugins to your gulp implementation at this to the beginning of your gulpfile.json.

var gulp = require('gulp')

, uglify = require('gulp-uglify')

, rename = require('gulp-rename')

, sourcemaps = require('gulp-sourcemaps')

, concat = require('gulp-concat');

Add a variable for your destination.

var dist = '/dist';

Syntax for creating a task

Gulp.task([task-name], function() {

Return gulp.src([glob-array]

.pipe([your-plugin])

.pipe([another-plugin])

.pipe(gulp.dest(dist));

});

## GULP Tutorial Part 4 – Creating an example application

TODO: Use a Yo or Yoemen Generator to create the MashupJS sample project.

TODO: For now use the steps outlined in the main ReadMe.md

## GULP TUTORIAL PART 5 – Sequence and Parallel task processing

The ability to execute Gulp task in sequence and parallel is still a moving target. By default Gulp leans toward executing all task in parallel because that is the more performant approach.

The option I chose was “run-sequence”. I chose this after battling with the other options. The Gulp 4 release should resolve many of the issues I struggled with.

From the command line install the **run-sequence** plugin

npm install run-sequence --save-dev

Add this module to your gulp modules list  
, runSequence = require('run-sequence')

All tasks created in this tutorial will have no other dependencies except the tasks we execute via the Gulp default task. The **runSequence** function will manage our dependencies.

One example is how we optimize JavaScript. Some JavaScript is found in \*.js files while other JavaScript is found after the compilation of TypeScript down to JavaScript. To minify and optimize JavaScript effectively it’s better to perform the JavaScript **Uglify** after the \*.ts, TypeScript, files have been transpiled. So a dependency exists between TypeScript and JavaScript.

We can accomplish this with **runSequence()**.

Here is our default task:

gulp.task('default', function() { runSequence('clean-dist',

'annotate',

'copy',

['coreservices', 'routeconfig', 'sass', 'tscompile', 'libs', 'grunt-merge-json:menu',

'tslint', 'jshint', 'minifyhtml', 'minifyimage'],

['uglifyalljs', 'minifycss'],

'watch'

);

});

Here is the sequence of execution

1 clean-dist

2 annotate

3 copy

4 (run in parallel) coreservices, routeconfig, sass, tscompile, libs, grunt-merge-json:menu, tslint, jshint, minifyhtml, minifyimage

5 uglifyalljs, minifycss

6 watch

These are all tasks you will have created by the end of this multi-part tutorial.

Notice the number of tasks executed in step 4. The more tasks you can run in parallel the faster your process will be. It’s important to optimize your process as much as possible so you can change a piece of code and immediately execute the optimized version without delay.

**Other Options**

Gulp 4.0 will have new methods series() and parallel(). This will be the preferred approach once released.

Orchestrator – is an NPM module that supports series and parallel processing.

## GULP TUTORIAL PART 6 – Handling errors with Plumber

Install the **gulp-plumber** plugin

npm install gulp-plumber --save-dev

Add the new plugins to your gulp required list

, plumber = require('gulp-plumber');

Add this to the top of your script file. Our **plumber** will use this function for logging errors to the console.

var onError = function(err) {

console.log(err);

};

Add the plumber function to each Gulp task.

// ---------------------------------------------------------------

// Watch specific tasks. This is to support the use of newer.

// ---------------------------------------------------------------

gulp.task('watch:annotate', function () {

return gulp.src(['src/index.controller.js', 'src/core/\*\*/\*.js', 'src/apps/\*\*/\*.js', '!src/core/lib/\*\*/\*', '!/\*\*/\*.min.js'], { base: 'src/./' })

**.pipe(plumber({**

**errorHandler: onError**

**}))**

.pipe(newer('src/./'))

.pipe(ngAnnotate())

.pipe(gulp.dest('src/./'));

});

The following tutorials will implement this **plumber** function with each task we create.

## GULP Tutorial Part 7 – Optimizing JavaScript/TypeScript

NOTE: Copy to ~~dist, Concat, uglify (compress and minify), rename, maps~~, JSHint, consider series/parallel

TODO: Add all client scripts to **bower** so that we aren’t copying unnecessary files to the **dist** directory.

In this part of the tutorial we are setting up our first gulp task.

### Annotation

Before we concatenate and minify let’s make sure our Angular code is in good shape. To make a long story short… Angular is based, largely, on the ability to directly inject dependencies. This ability is made possible because the name of the injected dependency is interpreted. As soon as a file is minified that name is change to ‘a’ or ‘b’ or whatever the next available small variable name is available. This breaks dependency injection.

Passing the dependency name as a string corrects this problem because static string names are not minified. You can do this yourself of let Gulp run a task to do this for you. Even if you decided to handle this while writing code it’s a good idea to run an annotation task anyway. Just in case.

From the command line install the **gulp-ng-annotate** plugin

npm install gulp-ng-annotate --save-dev

Add this module to your gulp modules list

var gulp = require('gulp')  
, annotate = require('gulp-ng-annotate')

Add the annotate task to your gulp file that will update the source code with good uri links.

gulp.task('annotate', function () {

return gulp.src(['src/index.controller.js', 'src/core/\*\*/\*.js', 'src/apps/\*\*/\*.js', '!src/core/lib/\*\*/\*', '!/\*\*/\*.min.js'], { base: 'src/./' })

.pipe(ngAnnotate())

.pipe(gulp.dest('src/./'));

});

The challenge presented by this task is updating the original files. Most of the time the original files will be left alone. In this case we can do the same but since annotation is actually a correction to code let’s just update our original files. To make this work we are using the { base: ‘src/./’ } option.

TIP: To keep your tasks running fast eliminate unnecessary processing by telling the task to ignore your JavaScript libraries. IE: '!src/core/lib/\*\*/\*'

More information on **gulp-ng-annotate** here.

<http://christian.fei.ninja/DRY-dependency-injection-in-Angular-with-gulp-ng-annotate/>

### Clean out ‘dist’

First, let’s clean out our ‘dist’ directory so we’re starting fresh. Execute the following from PowerShell.

npm install gulp-clean --save-dev

Add the following code to your gulpfile.js

var gulp = require('gulp')

, clean = require('gulp-clean');

gulp.task('clean', ['annotate'], function () {

return gulp.src('dist', { read: false })

.pipe(clean());

});

gulp.task('default', ['clean']);

From the command line you can execute the default task which includes cleaning out the ‘dist’ directory. If you don’t have files in your ‘dist’ directory or a ‘dist’ directory you can simply create the ‘dist’ directory and a few files for testing.

Until we’ve created a ‘dist’ directory you can execute the following lines at the command line. These will create the ‘dist’ directory and add a few files.

mkdir dist

ni dist/newjsfile.js -type file

ni dist/newjsfile2.js -type file

Now run the gulp command and watch the ‘dist’ directory disappear.

gulp

### Copy all src files to ‘dist’

We will keep our source (‘scr’) code separate from our distribution code so we don’t pollute our development environment. When executing the application we’ll set the index.html file of the ‘dist’ directory as the startup. This approach might seem to introduce challenges because when debugging you’ll need the ability to read the compressed and minified versions of JavaScript and CSS. Gulp will give us that ability.

Now that we’ve cleaned out the ‘dist’ directory in preparation for new files lets go ahead and copy all our source code to ‘dist’. Once we’ve copied the source code we can begin running tasks to optimize our code.

No new npm modules or gulp dependencies are required for the copy operation. Built in features, gulp.src and gulp.dest are enough.

Add the following new task to your gulpfile.js

gulp.task('copy', ['clean'], function () {

return gulp.src('src/\*\*/\*')

.pipe(gulp.dest('dist'))

});

**Task Dependencies**

Notice the [‘clean’] value in the second parameter. This is defining a dependency. Gulp defaults to the highest performance option, concurrency, so by default all tasks run in parallel. In this case the **copy** cannot occur until the **clean** operation is complete. Otherwise we’d be deleting files as we add them and that not very useful. The **copy** task will not begin until the **clean** task completes.

Execute the Gulp command and the dist directory will be deleted and rebuilt.]

Gulp

### concatination

Before we minify our JavaScript files let’s see if there are any files we wish to combine. We could simply minify all JavaScript files then concatenate them together but then we would lose the ability to map minified code to source code for debugging purposes. So we will concatenate together any code we desire and then execute a general minification task.

First install a concatenation plugin

npm install gulp-concat --save-dev

Add the new plugins to your gulp required list

var gulp = require('gulp')

, clean = require('gulp-clean')

, concat = require('gulp-concat');

Create a task that combines all the **core/common** files into one.

gulp.task('coreservices', ['copy'], function () {

return gulp.src('src/core/common/\*\*/\*')

.pipe(concat('core.services.js'))

.pipe(gulp.dest('./dist/'));

});

Create a task that combines all the **route.config**.js files together.

NOTE: This capability to combine the route config is what makes the drop in application style of MashupJS work.

gulp.task('routeconfig', ['copy'], function () {

return gulp.src(['src/core/config/route.config.js', 'src/apps/\*\*/route.config.js'])

.pipe(concat('route.config.js'))

.pipe(gulp.dest('./dist/'));

});

Now combine all your **bower libraries** together.

gulp.task('libs', ['clean'], function () {

return gulp.src(['bower\_components/\*\*/\*.js'])

.pipe(concat('libs.js'))

.pipe(gulp.dest('dist/core/lib/'));

});

Update your default task.

gulp.task('default', ['annotate', 'clean', 'copy', 'coreservices', 'routeconfig', 'libs']);

For additional documentation on gulp-concat: <https://github.com/wearefractal/gulp-concat>

### Compress and Minify JavaScript

It’s finally time to minify and compress JavaScript files. A normal solution would be to concatenate all \*.js files into a single file name, app.js, or something similar. The MashupJS is built to scale so large that a single \*.js file might become too large and lazy loading will be desired.

Here we will minify and compress individual JavaScript files and maps will be created for troubleshooting and debugging.

The challenge, here, is similar to what we encountered with the annotation task. We aren’t combining all of the JavaScripts into a single file. We will create minified and map files for each JavaScript file.

NOTE: This particular implementation is focused on scaling large enterprise applications so concatenating all JavaScript into a single file would cause a long initial load. Instead we will lazy load files as they are needed.

Install all the plugins required for this task. From the command line execute the following commands.

npm install gulp-rename --save-dev

npm install gulp-uglify --save-dev

npm install gulp-sourcemaps --save-dev

Add the new plugins to your gulp required list

var gulp = require('gulp')

, clean = require('gulp-clean')

, concat = require('gulp-concat')

, uglify = require('gulp-uglifyjs')

, rename = require('gulp-rename')

, sourcemaps = require('gulp-sourcemaps');

Add the new task to your gulpfile.js

gulp.task('uglifyalljs', ['copy', 'coreservices','routeconfig'], function () {

return gulp.src(['dist/\*\*/\*.js', '!/\*\*/\*.min.js', '!dist/core/lib/\*\*/\*', '!dist/core/common/\*\*/\*'], { base: 'dist/./' })

.pipe(sourcemaps.init())

.pipe(uglify())

.pipe(rename({

extname: '.min.js'

}))

.pipe(sourcemaps.write('./'))

.pipe(gulp.dest('dist/./'));

});

Update your default task.

gulp.task('default', ['annotate', 'clean', 'copy', 'coreservices', 'routeconfig', 'libs', 'uglifyalljs']);

## GULP Tutorial Part 8 – Optimizing CSS

NOTE: Copy to dist, Concat, uglify (compress and minify), rename, maps, linter for css, remove unused classes, consider series/parallel

NOTE: Get uglify working. After the app is up and running again add CSS remove unused.

TODO: Transpile from SASS here, then minify.

TODO: Make note that this is just minifying css. Later we’ll precompile SCSS down to CSS. This process will pick up where the SCSS process ends. The SCSS process can also minify and we might do it that way. I don’t know yet.

Install the **gulp-minify-css** plugin

npm install gulp-minify-css --save-dev

Add the new plugins to your gulp required list

, minifycss = require('gulp-minify-css');

Add the new task to your gulpfile.js

gulp.task('minifycss', ['copy'], function () {

return gulp.src(['dist/\*\*/\*.css', '!/\*\*/\*.min.css', '!dist/core/lib/\*\*/\*'], { base: 'dist/./' })

.pipe(sourcemaps.init())

.pipe(minifycss())

.pipe(rename({

extname: '.min.css'

}))

.pipe(sourcemaps.write('./'))

.pipe(gulp.dest('dist/./'));

});

Rather than concatenating css files we are simply minifying them in place and creating maps. This might change but for now the idea is the user can easily switch between Bootstrap themes and to make this possible the Bootstrap themes much be in separate files.

Later, when transpile from SASS we won’t need concatenation because the “@import” statement will pull multiple source files together for us.

For more information on gulp-minify-css.  
<https://www.npmjs.com/package/gulp-minify-css>

## GULP Tutorial Part 9 – Optimizing HTML

NOTE: copy to dist, uglify (compress and minify), rename, maps, linter for html, consider series/parallel

Install the **gulp-minify-html** plugin

npm install gulp-minify-html --save-dev

Add the new plugins to your gulp required list

, minifyhtml = require('gulp-minify-html');

Add the new task to your gulpfile.js

gulp.task('minifyhtml', ['copy'], function () {

return gulp.src(['dist/\*\*/\*.html', '!/\*\*/\*.min.html', '!dist/core/lib/\*\*/\*'], { base: 'dist/./' })

.pipe(sourcemaps.init())

.pipe(minifyhtml())

.pipe(rename({

extname: '.min.html'

}))

.pipe(sourcemaps.write('./'))

.pipe(gulp.dest('dist/./'));

});

Small templates won’t realize much improvement with html minification but every little big helps. Larger html files will benefit but while we’re at it lets just minify all html files.

For more information on gulp-minify-html.  
<https://www.npmjs.com/package/gulp-minify-html>

## GULP Tutorial Part 10 – Images

NOTE: copy to dist, optimize, rename, and consider series/parallel

Install the **gulp-imagemin** and **imagemin-pngquant** plugin. The gulp-imagemin comes with several optimizers for different image types but does very little for PNG files. The imagemin-pngquant will focus on optimizing PNG files.

I’ve decided to leave the css and other libraries alone but you might chose to optimize them as well. This demo is focused on optimizing our own images.

npm install gulp-imagemin --save-dev

npm install imagemin-pngquant --save-dev

Add the new plugins to your gulp required list

, imagemin = require('gulp- imagemin');

, pngquant = require('imagemin-pngquant')

Add the new task to your gulpfile.js

gulp.task('minifyimage', ['copy'], function () {

return gulp.src(['dist/\*\*/\*.{png,jpg,gif,ico}', '!dist/core/lib/\*\*/\*.\*', '!dist/core/css/\*\*/\*.\*'])

.pipe(imagemin({ progressive: true, optimizationLevel: 7, use: [pngquant()] }))

.pipe(gulp.dest('dist/./'));

});

For more information

<https://www.npmjs.com/package/gulp-imagemin>

<https://www.npmjs.com/package/imagemin-pngquant>

## GULP Tutorial Part 11 – JSON, calling grunt from gulp

Oh occasion you might need to combine JSON files. The MashupJS allows each app in the apps directory to define its own menu items. At build time we need these JSON files to be combined and saved with a specific file name so the menu.html template has access to all the menu items.

**Edge case calling for Grunt from Gulp**

So as it turns out our favorite plugins used in Grunt are not available in Gulp. Before using Gulp I use Grunt and a plugin named “grunt-merge-json” to combine separate JSON files. Not only did it combine the JSON but merged them and eliminated duplication. I tried several Gulp plugins but nothing worked as expected. In time a developer will build this plugin for Gulp but until then we can execute Grunt commands from our Gulp implementation.

**Install Grunt**

npm install -g grunt-cli

**Installing the plugin**

npm install grunt-merge-json --save-dev

**Create a Gruntfile**

Create a basic Gruntfile.json with the following content, in the root of your project.

module.exports = function(grunt) {

grunt.initConfig({

pkg: grunt.file.readJSON('package.json'),

"merge-json": {

menu: {

src: ['apps/\*\*/menu.json.txt'],

dest: 'dist/menu.json.txt',

},

},

});

grunt.loadNpmTasks('grunt-merge-json');

// task setup

grunt.registerTask('default', []);

grunt.registerTask('buildmenujson', ['merge-json:menu']);

};

**Install Gulp-grunt**

npm install gulp-grunt --save-dev

More information on gulp-grunt.

<https://www.npmjs.com/package/gulp-grunt>

**Add Grunt configuration to Gulp**

// -------------------------------------------------

// Grunt configuration

require('gulp-grunt')(gulp, {

// These are the default options but included here for readability.

base: null,

prefix: 'grunt-',

verbose: false

});

// -------------------------------------------------

These are default configurations. The base represents the path to the Gruntfile.json. Because the Gruntfile.json is in the root directory base: can be null.

We can now call the grunt task the same way we would a Gulp task but with the prefix “grunt=”.

**Test Grunt Plugin**

Gulp grunt-merge-json:menu

**Add task to gulp**

One you are satisfied this is working you can add it to your gulp default task.

gulp.task('default', ['annotate', 'clean', 'copy', 'coreservices', 'routeconfig', 'libs'

, 'uglifyalljs', 'minifycss', 'minifyhtml', **'grunt-merge-json:menu'**]);

~~Install the~~ **~~gulp-extend~~** ~~plugin~~

~~npm install gulp-extend --save-dev~~

~~Add the new plugins to your gulp required list~~

~~, extendJSON = require('gulp-extend');~~

~~Add the new task to your gulpfile.js~~

~~gulp.task('minifyhtml', ['copy'], function () {~~

~~return gulp.src(['dist/\*\*/\*.html', '!/\*\*/\*.min.html', '!dist/core/lib/\*\*/\*'], { base: 'dist/./' })~~

~~.pipe(sourcemaps.init())~~

~~.pipe(minifyhtml())~~

~~.pipe(rename({~~

~~extname: '.min.html'~~

~~}))~~

~~.pipe(sourcemaps.write('./'))~~

~~.pipe(gulp.dest('dist/./'));~~

~~});~~

## GULP Tutorial Part 12 – JSHINT

TODO: Go through Grunts jshint config and merge in anything I like.

TODO: Walk the user through the config and show some of the options available.

JSHint is a static analysis tool used to analyze JavaScript code for quality and standards/style enforcement. The example below demonstrates how to analyze your code but it can also be piped into transpilation of code from TypeScript to JavaScript.

Install the **gulp-jshint**

npm install gulp-jshint --save-dev

npm install jshint-stylish --save-dev

npm install gulp-jshint-html-reporter --save-dev

Add the new plugins to your gulp required list

, jshint = require('gulp-jshint')

, stylish = require('jshint-stylish')

, jshintfileoutput = require('gulp-jshint-html-reporter')

Add the new task to your gulpfile.js

gulp.task('jshint', ['copy'], function () {

return gulp.src(['./dist/\*\*/\*.js', '!dist/core/lib/\*\*/\*.\*', '!\*\*/\*.min.js', '!dist/core/css/\*\*/\*.\*'])

.pipe(jshint('.jshintrc'))

.pipe(jshint.reporter(stylish))

.pipe(jshint.reporter('gulp-jshint-html-reporter', { filename: 'jshint-output.html' }))

;

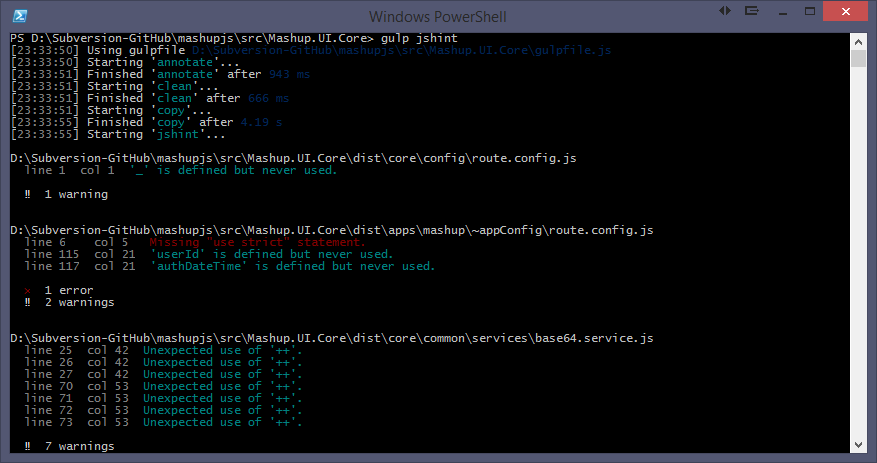
});

**Execute task**

gulp jshint

**Using JSHint reporters**

Reporters receive feedback from JSHint and format it into something human readable. By default you’ll get raw text from JSHint and it will be displayed at the command line. Adding the ‘jshint-stylish’ plugin and passing it to the jshint.reporter(stylish) gives you a more readable output to the command line.

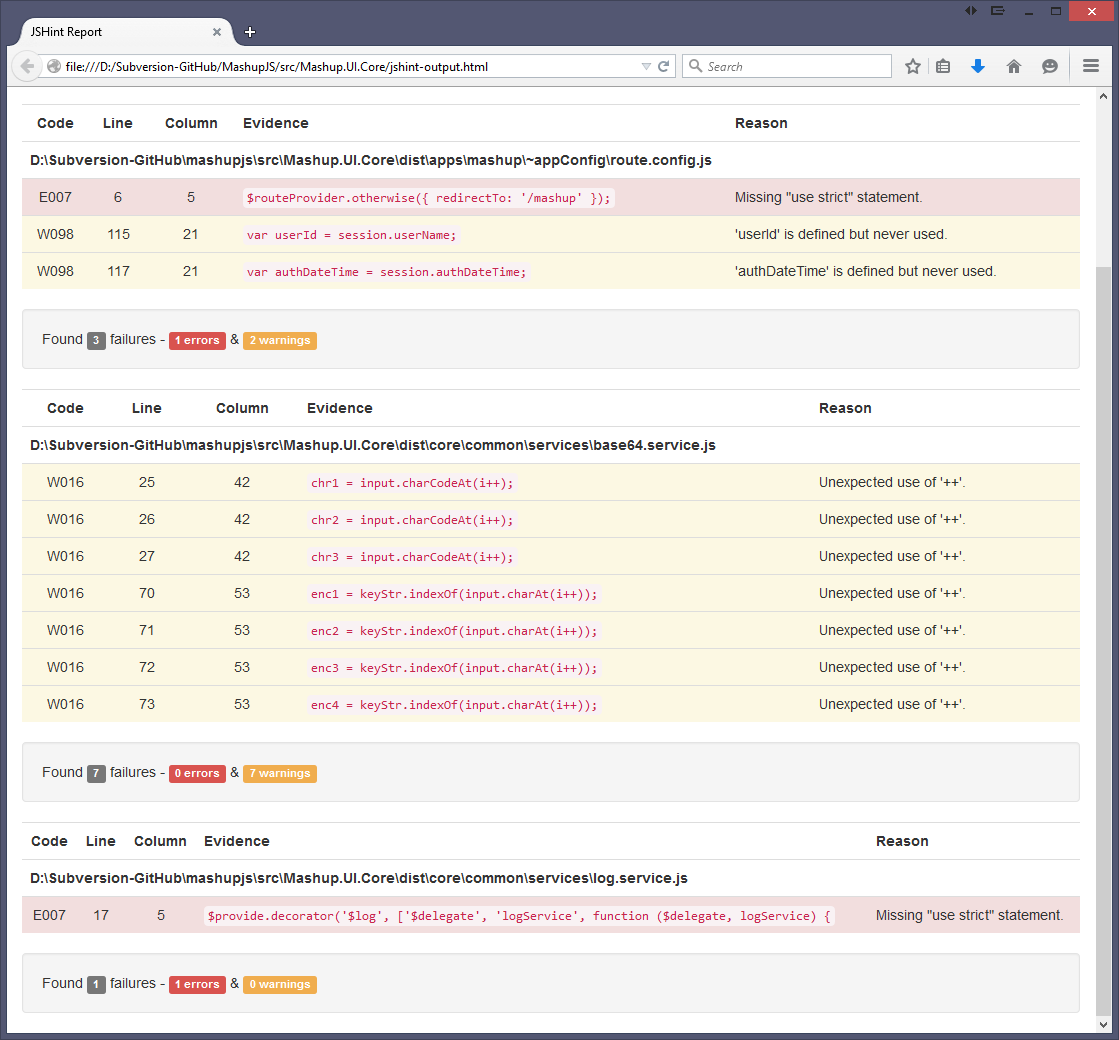


Taking this concept a step further, the ‘gulp-jshint-html-reporter’ generates an html document, as the name of the plugin implies.

For more information. There isn’t much in the way of documentation but the source code is available and you have access to the author here.

<https://www.npmjs.com/package/gulp-jshint-html-reporter>

<https://github.com/ivan-vesely/gulp-jshint-html-reporter>



**Creating a** **.jshintrc file**

You can customize your own jshint rules in a file called .jshintrc. I’m starting with options recommended by John Papa.

<https://github.com/johnpapa/angular-styleguide#style-y230>

**.jshintrc TIP**

You cannot easily create files beginning with a dot in windows.

Steps to create the .jshintrc file

* Create a file named “.jsintric.” Notice the ending dot.
* Windows will prompt you to confirm the extension.
* Finally windows simply removes the ending dot for you. Now you have the file “.jsintric”.

**Options List**

<http://jshint.com/docs/options/>

For more information

<https://www.npmjs.com/package/gulp-jshint>

## GULP Tutorial Part 13 – TypeScript

TypeScript is a language used to build large scale JavaScript applications. TypeScript code is transpiled (note: this is a word) down to ES5 JavaScript. Angular 2.0 is written in TypeScript. ES2015/ES2016, formerly ES6/ES7, features are available in TypeScript.

Links for installing TypeScript, for your version of Visual Studio, are here.  
<http://www.typescriptlang.org/>

Dan Wahlin’s take on TypeScript in a Gulp workflow  
<http://weblogs.asp.net/dwahlin/creating-a-typescript-workflow-with-gulp>

**Create a TypeScript file work with**

The MashupJS isn’t using TypeScript yet so we will need to create a TypeScript file for this example.

Create a file in the root of the scr folder named myTypeScript.ts

Add the following code snippet from <http://www.typescriptlang.org/Playground> to the new file.

class Greeter {

greeting: string;

constructor(message: string) {

this.greeting = message;

}

greet() {

return "Hello, " + this.greeting;

}

}

var greeter = new Greeter("world");

var button = document.createElement('button');

button.textContent = "Say Hello";

button.onclick = function() {

alert(greeter.greet());

}

document.body.appendChild(button);

**Expected Result**

Once we’ve transpiled the TypeScript down to JavaScript (ES5), it should be a minified version of the following.

var Greeter = (function () {

function Greeter(message) {

this.greeting = message;

}

Greeter.prototype.greet = function () {

return "Hello, " + this.greeting;

};

return Greeter;

})();

var greeter = new Greeter("world");

var button = document.createElement('button');

button.textContent = "Say Hello";

button.onclick = function () {

alert(greeter.greet());

};

document.body.appendChild(button);

**Installing plugins**

npm install gulp-typescript --save-dev

npm install gulp-tslint --save-dev

npm install gulp-tslint-stylish --save-dev

**Add the new plugins to your gulp required list**

, tsc = require('gulp-typescript')

, tslint = require('gulp-tslint')

, tsstylish = require('gulp-tslint-stylish')

**Add the new task to your gulpfile.js**

gulp.task('tscompile', ['copy'], function () {

return gulp.src(['./dist/\*\*/\*.ts', '!dist/core/lib/\*\*/\*.\*', '!dist/core/css/\*\*/\*.\*'])

.pipe(sourcemaps.init())

.pipe(ts({

target: 'ES5',

declarationFiles: false,

noExternalResolve: true

}))

// Exporting the ES5 .js file. This is never used so you can remove the following two lines.

// You might want to keep them so you can evaluate how TypeScript is transpiling your JavaScript.

// This also gives JSHint a shot at linting the JavaScript version of your TypeScript code.

.pipe(rename({ extname: '.js' }))

.pipe(gulp.dest('dist/./'))

// Creating the optimized JavaScript file.

.pipe(uglify())

.pipe(rename({ extname: '.min.js' }))

.pipe(sourcemaps.write('./'))

.pipe(gulp.dest('dist/./'));

});

**Executing the task**

You can execute the task individually

gulp tscompile

**And add the new task to your default task**

gulp.task('default', ['annotate', 'clean', 'copy', 'coreservices', 'routeconfig', 'libs'

, 'uglifyalljs', 'minifycss', 'minifyhtml', 'grunt-merge-json:menu', 'minifyimage'

**, 'tscompile'**, 'jshint']);

**How it works**

First the task transpiles the TypeScript code down to ES5, ECMAScript 5.

Then the new ES5 JavaScript is emitted. This file will not be used by the min.js.map. The map file will point directly to the TypeScript, “.ts”, file. The reason we are emitting the “.js” is so the TypeScript code can participate in the JSHint process. You’ll notice the ‘jshint’ task now has a dependency on the ‘tscompile’ task.

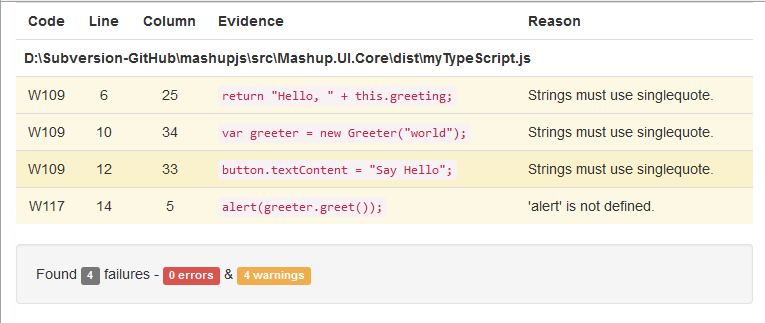
gulp.task('jshint', ['copy'**, 'tscompile'**], function () {

Finally the TypeScript files is emitted as a minified JavaScript file with an associated map file linking it back to the TypeScript file.

**jshint-output.html**

If you run the new task with gulp tscompile you’ll notice the jshint-output.html file has been updated.

If you double-click this file from explorer.exe then you’ll notice our JavaScript, emitted by TypeScript, has a few issues.



**Fixing the JSHint warnings**

These are pretty simple changes to make. Replace all the double quotes with single quotes and define the alert() method and JSHint is satisfied. The end result should be this.

**/\*global alert \*/**

class Greeter {

greeting: string;

constructor(message: string) {

this.greeting = message;

}

greet() {

return 'Hello, ' + this.greeting;

}

}

var greeter = new Greeter('world');

var button = document.createElement('button');

button.textContent = 'Say Hello';

button.onclick = function () {

alert(greeter.greet());

}

document.body.appendChild(button);

**Linting TypeScript**

TypeScript has language constructs ES5 JavaScript does next so automating a TypeScript specific linter might seem redundant. It’s not.

Add the following task to your gulpfile.js

gulp.task('tslint', ['copy'], function () {

return gulp.src(['./dist/\*\*/\*.ts', '!dist/core/lib/\*\*/\*.\*', '!dist/core/css/\*\*/\*.\*'])

.pipe(tslint())

.pipe(tslint.report('verbose', {

emitError: false,

sort: true,

bell: true

}))

});

**TSLint reporter**

I was unable to find a plugin to export tslint errors to an html file as we did for JSHint. If this plugin becomes available I’ll add it to this post here.

**TSLint Configuration**

Configuration information for TSLint is stored in a file named “tslint.json”.

Here is a good starting point. I’ve borrowed this from other develop posts and it seems to be a good list.

Add the following to your tslint.json file unless you decide to use what comes with the MashupJS.

{

"rules": {

"class-name": true,

"curly": true,

"eofline": false,

"forin": true,

"indent": [true, 4],

"label-position": true,

"label-undefined": true,

"max-line-length": [true, 140],

"no-arg": true,

"no-bitwise": true,

"no-console": [true,

"debug",

"info",

"time",

"timeEnd",

"trace"

],

"no-construct": true,

"no-debugger": true,

"no-duplicate-key": true,

"no-duplicate-variable": true,

"no-empty": true,

"no-eval": true,

"no-imports": true,

"no-string-literal": false,

"no-trailing-comma": true,

"no-trailing-whitespace": true,

"no-unused-variable": false,

"no-unreachable": true,

"no-use-before-declare": true,

"one-line": [true,

"check-open-brace",

"check-catch",

"check-else",

"check-whitespace"

],

"quotemark": [true, "single"],

"radix": true,

"semicolon": true,

"triple-equals": [true, "allow-null-check"],

"variable-name": false,

"whitespace": [true,

"check-branch",

"check-decl",

"check-operator",

"check-separator"

]

}

}

For more information

<https://www.npmjs.com/package/gulp-typescript>

## GULP Tutorial Part 14 – SASS

For your SASS implementation we will use the gulp-sass plugin.

[Why use SASS over CSS]

Install the **gulp-sass** plugin

npm install gulp-sass --save-dev

Add the new plugins to your gulp required list

, sass = require('gulp-sass')

Add the new task to your gulpfile.js

gulp.task('sass', ['copy'], function () {

gulp.src('./dist/\*\*/\*.scss', { base: 'dist/./' })

.pipe(sass())

// Catch any SCSS errors and prevent them from crashing gulp

.on('error', function (error) {

console.error(error);

this.emit('end');

})

.pipe(gulp.dest('dist/./'));

});

Learn more about SASS here  
<http://sass-lang.com/>

## GULP Tutorial Part 16 – Watch

NOTE: watch newer only, make not of the performance over Grunt, and consider series/parallel

Running the default task for Gulp, with all our tasks included, will consume more CPU and time than is required. For dealing with files as they change we can use the Watch plugin.

When the watch is triggered it executes whatever tasks you have assigned. This still consume more CPU and time than is required. If you change one JavaScript file then all of your JavaScript do not need reprocessed.

Adding “Newer” to “Watch” gives us a good combination.

Install the **gulp-watch** and **Newer**

npm install gulp-watch --save-dev

npm install gulp-newer --save-dev

Add the new plugins to your gulp required list

, watch = require('gulp-watch')

, newer = require('gulp-newer')

Add the new task to your gulpfile.js

TODO: Change the code below and transfer my notes to the new code.

Note: Notice the .pipe(newer(imgSrc)

* What this is doing is comparing the src with the dest to see what is actually new.

// Minify any new images

gulp.task('images', function() {

// Add the newer pipe to pass through newer images only

return gulp.src(imgSrc)

.pipe(newer(imgDest))

.pipe(imagemin())

.pipe(gulp.dest(imgDest));

});

**Execute task**

gulp jshint

## GULP TUTORIAL PART 17 - LiveReload

## GULP Tutorial Part 18 – ES6

## GULP Tutorial Part 19 – Initiate tests

## GULP Tutorial Part 20 – Useful Gulp Commands & Tips

**Installing gulp**

Execute both of these. The first adds gulp locally so it can be used by npm. The second installs gulp globally so it can be accessed from the command line.

npm install gulp --save-dev

npm install gulp -g

**Retrieve gulp Version**

Grunt -version

**Installing plugins**

The syntax for Grunt plugins is

install [plugin-name] --save-dev

For example if you want to minify and concatenate your JavaScript for performance you would install two plugins.

Perform a quick google search and you’ll find this site

<https://github.com/gruntjs/grunt-contrib-uglify>

npm install grunt-contrib-uglify --save-dev

Perform a quick google search and you’ll find this site

<https://github.com/gruntjs/grunt-contrib-concat>

npm install grunt-contrib-concat --save-dev

**Retrieve Gulp version**

Gulp --v

**Every Gulp file needs a default task. To execute Gulp’s default task**

grunt

**It’s useful to run specific tasks that you have configured**

grunt [task-name]

**Get a list of grunt commands**

Grunt –help

**To verify a plugin is not blacklisted**

Gulp --verify

**Testing tasks while building your gulpfile.js**

You can type gulp [task-name] and your task will run. If it has any dependencies then those dependencies will run first.

gulp [task-name]

## GULP Tutorial Part 21 – Glob Tips

“dir/\*” – includes all files

“dir/\*\*” – includes all files and directories

<https://github.com/isaacs/node-glob>

<http://mywiki.wooledge.org/glob#preview>

# GULP TUTORIAL PART ? – Useful NPM Packages/Commands

## Commands Cheat Sheet

* Find outdated modules
  + npm outdated -–depth=0
  + npm outdated –-json -–depth=0
* Updating a package
  + npm install grunt-contrib-uglify@\* --save-dev

## PowerShell (primer)

Windows users can use either the Command Prompt or PowerShell.

PowerShell is pre-installed on Windows 8 or newer installations.

From Start: *Search programs and files* type “powershell”. Select “powershell.exe”.



### Syntax

Command Syntax: **application action –flags arguments**

For help with any application add the **–h** or **–help** flags for additional instructions.

The **tab** key autocompletes your statement.

### Adding and removing files

To create a new item use the **ni** command. This might not seem useful with Visual Studio 2013 because any file added must also be added to your project file. Visual Studio 2015 does nto have a project file needing updates. Instead a Glob pattern is used to determine what files should and should not be included in the project. That being the case, suddenly, **ni** makes more sense.

Example: ni newjsfile.js -type file

Or : new-item newjsfile.js –type file

Removing files: ri newjsfile.js or remove-item newjsfile.js

## Installing NodeJS and NPM Packages

Install NodeJS from:

<https://nodejs.org/>

<https://iojs.org/en/index.html>

Install NPM packages with the following syntax

npm install [global option –g] [package-name] [options]

Example: (You need to install gulp both locally and globally)

npm install gulp --save-dev

npm install gulp -g

More information can be found in the TODO: Link to Gulp Tutorial: Setting up Gulp

<https://docs.npmjs.com/getting-started/installing-npm-packages-locally>

## Version updates

**How to configure for version updates**

There are multiple options for keeping NPM packages up to date. The approach you choose might depend on your development workflow and automated testing solution. IE: If you have good automated testing it might be safe to allow the latest versions. If not then you might want to choose a more deliberate approach to npm versioning.

## Version updates: Option 1 – Using node tools

**Check to see which npm packages are out of date**

Display colored rows

npm outdated -–depth=0

Display in json which includes current, wanted, latest version numbers

npm outdated –-json -–depth=0



Note: Not all your packages will be displayed. Only the outdated packages.

Note: If you modify the command to include “-g” then you’ll get a list of your outdated global packages.

**To update packages one at a time**

npm install [package-name]@\* [save?]

npm install grunt-contrib-uglify@\* --save-dev

## Version updates: Option 2 – Using npm-check-updates

Using the npm-check-updates package you can keep all your packages updated.

<https://www.npmjs.com/package/npm-check-updates>

npm install -g npm-check-updates

Then execute the following command to see what packages can be updated.

npm-check-updates



To upgrade all your packages

npm-check-updates –u [-g option for global packages]

Now your package.json is updated.

Then execute an npm install to update the package installations.

npm install [-g option for global packages]

## npm versioning semantics

<https://docs.npmjs.com/misc/semver>

<http://semver.org/>

# MashupJS gulp Implementation

The Mashup will maintain both Grunt and Gulp. You can use whichever works best for your workflow.

Grunt is a task runner used in the development and the build processes. All files are processed and distributed to the “/dist” directory. Everything needed for deployment should be found here.

To make this work there are a few things you need to do. If you are using VS2015 then some of this will happen auto-magically.

This document will walk you through the basic setup and usage of Grunt, how Grunt is configured for the Mashup, and how to use Grunt as part of the Mashup workflow.

TODO: Explain Mashup’s Grunt implementation