Gulp

Introduction

Gulp is a build automation tool for Javascript. It's similar to Ant for Java in that you have to specify tasks and task dependencies. When you run a task, it'll make sure all the dependent tasks are run first.

There are other Javascript build automation tools out there (e.g. grunt and webpack). The thing that makes gulp unique is that the API is very simple and modifications/transformations are done as a stream. You define what a stream does, then files go through that stream and have your modifications/transformation applied, then those files get written out to some destination.

The 4 main APIs of gulp are…

- gulp.task() → define a new task
- gulp.src() → start a new stream
- gulp.dest() → write out files to the stream
- gulp.watch() → continually watch and re-run a task everytime something changes

Here's a simple example of a gulp task that checks a JS files against a linter/stylechecker…

```
gulp.task('vet', function() {
    return gulp
        .src('**/*.js')
        .pipe(gulp-if(args.verbose, $.print()))
        .pipe(gulp-jscs())
        .pipe(gulp-jshint())
        .pipe(gulp-jshint().reporter('jshint-stylish', {verbose: true}))
        .pipe(gulp-jshint().reporter('fail'));
});
```

Environment Setup

You can install Gulp through NodeJS's package manager (called NPM).

To install NodeJS, check out https://nodejs.org/en/download/package-manager/.

To install Gulp…

```
~ $ sudo npm install -g gulp
~ $ sudo npm install -g bower
```

NOTE: What's -g for? Check out https://stackoverflow.com/a/8951576/1196226. The idea is that -g should be used to install things globally like commandline tools.

NOTE: What's bower for? bower is like npm, but it's for client-side packages (stuff that runs in the browser?). We're going to use bower to pull in stuff like angular-js for our webapp. npm is mostly used for server-side packages.

Once that's done, you can run bower and gulp as a commandline tools…

```
~ $ gulp -v
[19:41:01] CLI version 3.9.1
~ $ bower -v
1.8.2
```

Project Setup

There are 3 things you need to do to setup a new project: create a NodeJS package for your project, create a gulp build script, and install any plugins you want to use with gulp.

NodeJS Package

To begin, create and enter the directory where your project should exist…

```
~ $ mkdir myproject
~ $ cd myproject/
~/myproject $
```

Once that's done, you need to create a package.json file that will be used to define your project as a nodejs package…

```
~/myproject $ echo {} > package.json
```

NOTE: We're creating an empty json file, which works, but I think the standard way to do this is by running "npm init".

Once that's done, you need to install gulp locally for your project/package. To do this…

```
~/myproject $ npm install gulp --save-dev
```

NOTE: Note that we don't have a -g here like we did in the environment setup section.

Notice the --save-dev flag. That flag will put in gulp as a devDependency. A devDependency is a dependency required for development of your the development of your nodeJS package...

```
~/myproject $ cat package.json
{
  "devDependencies": {
    "gulp": "^3.9.1"
  }
}
```

NOTE: If you want to add in a dependency as a runtime dependency, you can use --dev, which will add the package under dependency instead of devDependency.

Gulp Build Script

The final thing you need to do is create the gulpfile.js file which will contain all of your build automation logic. This is equivalent to Ant's build.xml. You specify your tasks, task dependencies, etc.. all in this file.

Here's an example gulpfile.js with a single task…

```
var gulp = require('gulp'); // nodejs equiv of import

gulp.task('my-task', function() {
    console.log('My task!!!');
})
```

And here's how to run that task…

```
~/myproject $ gulp my-task
[20:19:23] Using gulpfile ~/pluralsight-gulp/gulpfile.js
[20:19:23] Starting 'my-task'...
My task!!!
[20:19:23] Finished 'my-task' after 99 µs
```

Gulp Plugins

For most build scripts, you'll typically want to use plugins. You can grab plugins directly from NodeJS. For example, gulp-prettier is a plugin that you can use to prettify source code.

Here's how to get it…

```
~/myproject $ npm install gulp-prettier --save-dev
```

NOTE: Remember that --save-dev will save it in your nodeJS package manifest (package.json) as a development dependency.

You can then import it into your code just like anything else…

```
var gulp = require('gulp');
var gulp_prettier = require('gulp-prettier');

gulp.task("prettify", () =>
  gulp
```

```
    .src("src/**/*.js")
    .pipe(
      gulp_prettier({
        singleQuote: true,
        trailingComma: "all"
      })
    )
    .pipe(gulp.dest(file => file.base))
);
```

You'll typically use MANY gulp plugins, so it'll become tedious to go through and import them individually. There's a plugin called gulp-load-plugins that will find all the gulp plugins you've installed and present them to you in a single object. Check out https://www.npmjs.com/package/gulp-load-plugins for information on how to set it up.

API

The fundamental units within gulp are tasks and streams. You create a dependency graph of tasks, and each of those tasks generates a stream of things to do to some files.

The 4 core APIs of gulp are...

- gulp.task() → define a new task
- gulp.src() → start a new stream
- gulp.dest() → write out files to the stream
- gulp.watch() → continually watch and re-run a task everytime something changes

Tasks

At the top-level you define a task using gulp.task(). What that task does is up to you, but the key thing to remember about tasks is that they can have dependencies on other tasks.

For example, imagine we had a task that depends on 2 other tasks…

```
var gulp = require('gulp');

gulp.task('parent-task1', function() {
    console.log("parent task1");
});

gulp.task('parent-task2', function() {
    console.log("parent task2");
});
```

```
gulp.task('child-task', ['parent-task1', 'parent-task2'], function() {
    console.log("child task");
});
```

When we try to run that task, we see that it automatically runs the 2 tasks it depends on first...

```
~/myproject $ gulp child-task
[19:33:56] Using gulpfile ~/myproject/gulpfile.js
[19:33:56] Starting 'parent-task1'...
parent task1
[19:33:56] Finished 'parent-task1' after 105 μs
[19:33:56] Starting 'parent-task2'...
parent task2
[19:33:56] Finished 'parent-task2' after 44 μs
[19:33:56] Starting 'child-task'...
child task
[19:33:56] Finished 'child-task' after 48 μs
```

Streams

Rather than having raw javascript, our tasks will almost always define streams.

Streams start off with gulp.src(), which takes in a glob or regex expression (or an array of) that define what files the stream should take in. You can then chain on calls to .pipe() which will define tasks to perform on that stream. Finally, you can chain on a call to .dest() which will write out the files to some path.

For example, here's a gulp task creates a stream which concatenates all your javascript files together…

```
gulp.task('minify-js', function () {
      return gulp.src('./js/**/*.js') // path to your files
          .pipe(uglify())
          .pipe(gulp.dest('path/to/destination'));
});
```

NOTE: uglify() is from the plugin gulp-uglify. You'll learn about plugins in the plugins section.

Note that we're RETURNING the stream we're creating. We need to do this so any tasks that depend on this task know when this task finishes. If you want this task to run asynchronously (which you almost never would,

Watches

You can also set a task to watch the filesystem for changes to certain files. If changed, the tasks you specify will get run.

For example, here's a task that triggers a watch instead of a stream…

```
gulp.task('watch', function() {
    gulp.watch('source/javascript/**/*.js', ['minify-js']);
});
```

NOTE: You aren't limited to 1 watch. You can have multiple watches.

Now if we run this task, the gulp process will never exit. Instead it'll just sit idle until there's a change in the files we specified (source/javascript/**/*.js) and run the tasks we defined (minify-js)...

Best Practices

Configuration File

For larger projects, it's typical to have a gulp.config.js file that you import inside your gulpfile.js. This configuration file would have common things that you use/re-use throughout your gulpfile.

For example, common paths/globs/strings/settings would get placed inside this gulp.config.js file…

```
module.exports = function() {
    var baseSrc = 'src';

    var config = {
        jsPath: baseSrc + '/js',
        htmlPath: baseSrc + '/html',
        targetPath: 'target'
    };

    return config;
};
```

Then, in your gulpfile.js, import your config file…

```
var gulp = require('gulp');
var gplugins = require('gulp-load-plugins')({lazy: true});
var config = require('./gulp.config')();
```

> **NOTE**: What's with the extra () at the end? Remember that our gulp.config.js is exporting a function. As such, we need to run that function when we import it so it returns the actual config object.

Clean Task (del)

Cleaning is when you delete your build files (because you want to do a fresh build). There is no gulp specific way to do this. Remember that gulp streams process files and dump the outputs somewhere. We aren't processing anything here.

The gulp github page (https://github.com/gulpjs/gulp/blob/master/docs/recipes/delete-files-folder.md) recommends using the del node module to clean files. Apparently the del module supports glob patterns so it will work well with whatever existing glob constants you've made for your build script.

Example…

```
~/myproject $ npm install del --save-dev
```

```
var gulp = require('gulp');
var del = require('del');

gulp.task('clean:mobile', function () {
  return del([
    'dist/report.csv',
    // here we use a globbing pattern to match everything inside the
`mobile` folder
    'dist/mobile/**/*',
    // we don't want to clean this file though so we negate the pattern
    '!dist/mobile/deploy.json'
  ]);
});
```

Listing Tasks (gulp-task-listing)

This plugin allows you to dump out a friendly view of your list of gulp tasks.

To install…

```
~/myproject $ npm install --save-dev gulp-task-listing
```

Usage example…

```
var gulp = require('gulp');
var gulptasklisting = require('gulp-task-listing');

...

gulp.task('help', function() { gulptasklisting(); });
gulp.task('default', ['help']);
```

Now if you run the the help task (or just gulp by itself without any tasks), gulp will give you a friendly view of all the tasks you can run.

Logging (gulp-util)

Gulp-util is a plugin with a bunch of handy utility methods that you can use with gulp. The most import functionality is essentially colored logging, templating, and giving you the ability to do conditional operations in streams (gulp-if is also available for this).

To install…

```
~/myproject $ npm install --save-dev gulp-util
```

Usage example…

```
var gulp = require('gulp');
var gutil = require('gulp-util');

gutil.log('stuff happened', 'Really it did', gutil.colors.magenta('123'));

gutil.replaceExtension('file.coffee', '.js'); // file.js

var opt = {
  name: 'tod',
  file: someGulpFile
};
gutil.template('test <%= name %> <%= file.path %>', opt) // test tod /hi.js


// gulp should be called like this :
// $ gulp --type production
gulp.task('scripts', function() {
  return gulp.src('src/**/*.js')
```

```
    .pipe(concat('script.js'))
    .pipe(gutil.env.type === 'production' ? uglify() : gutil.noop())
    .pipe(gulp.dest('dist/'));
});
```

> **NOTE**: gutil.noop() is essentially letting us perform some action if a certain condition is met. There's also a plugin called gulp-if that does the same thing.

Verbosity (gulp-print)

Gulp-print is a plugin that you can pass into .pipe() to have it show the files it's working on when you run the task. This is great if you want to make sure the files being brought in via a glob are actually being brought in.

> **NOTE**: use this plugin with an arguments library like yargs, such that if a user passes in --verbose, gulp-print() will be activated. gulp-util has a function called gutil.noop() that can help with this: .pipe(yargs.argv.verbose ? gprint() : gutil.noop()).

To install…

```
~/myproject $ npm install --save-dev gulp-print
```

Usage example…

```
var gulp = require('gulp');
var gprint = require('gulp-util');

gulp.task('test', function() {
  return gulp.src('src/**/*.js')
    .pipe(gprint())
    .pipe(...);
});
```

Style Checking (gulp-eslint)

ESLint is a linter/checkstyler for EMCAscript/Javascript. One thing to keep in mind is that a lot of editors support ESLint as well. For example, if you configure your project to use ESLint through a .eslintrc config file (discussed later on), brackets will pick that up and use those settings.

To install…

```
~/myproject $ npm install --save-dev gulp-eslint
```

Before you can use the plugin, you need to configure it. For a detailed configuration guide, check out https://eslint.org/docs/user-guide/configuring, but here is a common .eslintrc file that you can put in the root of your project…

```
{
    "extends": ["eslint:recommended"],
    "env": {
        "browser": true,
        "node": true
    }
}
```

NOTE: This uses recommended ES6 rule settings and makes it so that browser globals (e.g. document) or node globals (e.g. require?) don't break the linter.

Usage example…

```
var gulp = require('gulp');
var eslint = require('gulp-eslint');

gulp.task('lint', function() {
    // ESLint ignores files with "node_modules" paths.
    // So, it's best to have gulp ignore the directory as well.
    // Also, Be sure to return the stream from the task;
    // Otherwise, the task may end before the stream has finished.
    return gulp.src(['**/*.js','!node_modules/**'])
        // eslint() attaches the lint output to the "eslint" property
        // of the file object so it can be used by other modules.
        .pipe(eslint())
        // eslint.format() outputs the lint results to the console.
        // Alternatively use eslint.formatEach() (see Docs).
        .pipe(eslint.format())
        // To have the process exit with an error code (1) on
        // lint error, return the stream and pipe to failAfterError last.
        .pipe(eslint.failAfterError());
});
```

Error Handling (gulp-plumber)

Sometimes a plugin that you use may produce an error in your stream, which may break the stream. If this happens, sometimes gulp will fail silently. The gulp-plumber plugin helps guard against this issue.

To install…

```
~/myproject $ npm install gulp-plumber --save-dev
```

Usage example…

```
var plumber = require('gulp-plumber');
var coffee = require('gulp-coffee');

gulp.src('./src/*.ext')
    .pipe(plumber()) // must be added first?
    .pipe(coffee())
    .pipe(gulp.dest('./dist'));
```

> **NOTE**: There's also another tool for error handling called pump… https://github.com/terinjokes/gulp-uglify/blob/master/docs/why-use-pump/README.md#why-use-pump.

Simplified Plugin Loading (gulp-load-plugins)

Your gulp setup will typically make use of a ton of different plugins. Instead of having to import each of those plugins directly, you can use the gulp-load-plugins to load up all the plugins you have installed.

To install…

```
~/myproject $ npm install --save-dev gulp-load-plugins
```

Usage example…

```
var gulp = require('gulp');
var gplugins = require('gulp-load-plugins')({lazy: true});
//var eslint = require('gulp-eslint');
//var gprint = require('gulp-print');


gulp.task('lint', function() {
    // ESLint ignores files with "node_modules" paths.
    // So, it's best to have gulp ignore the directory as well.
    // Also, Be sure to return the stream from the task;
    // Otherwise, the task may end before the stream has finished.
    return gulp.src(['**/*.js','!node_modules/**'])
        // eslint() attaches the lint output to the "eslint" property
        // of the file object so it can be used by other modules.
        .pipe(gplugins.print())
        .pipe(gplugins.eslint())
        // eslint.format() outputs the lint results to the console.
        // Alternatively use eslint.formatEach() (see Docs).
        .pipe(gplugins.eslint.format())
        // To have the process exit with an error code (1) on
        // lint error, return the stream and pipe to failAfterError last.
        .pipe(gplugins.eslint.failAfterError());
});
```

Notice what's happening in the example. We only import gulp-load-plugins, and gulp-load-plugins gives us access to all the plugins we have installed. By convention, it strips out the gulp- prefix from the start of the plugin name and provides a function with that name to access the plugin. So for example, to access the gulp-eslint plugin all you need to do is gulp-load-plugins.eslint.

CSS Preprocessors

SASS (gulp-sass)

SASS is a CSS preprocessor that makes writing CSS a bit less of a pain and works around some browser quirks (e.g. vendor prefixes). Use this plugin to compile your SCSS files to CSS.

To install…

```
~/myproject $ npm install gulp-sass --save-dev
```

Usage example…

```
var gulp = require('gulp');
var sass = require('gulp-sass');

gulp.task('sass', function () {
  return gulp.src('./sass/**/*.scss')
    .pipe(sass().on('error', sass.logError))
    .pipe(gulp.dest('./css'));
});

gulp.task('sass:watch', function () {
  gulp.watch('./sass/**/*.scss', ['sass']);
});
```

Production and Optimizations

Injecting your CSS/JS into HTML (gulp-inject)

If you want to automatically insert your js and css files into your html files, you would use the gulp-inject plugin.

To install…

```
~/myproject $ npm install --save-dev gulp-inject
```

Usage example…

```
var gulp = require('gulp');
var gulpinject = require('gulp-inject')

gulp.task('inject', function () {
    var files_to_inject = gulp.src(
        [
            './bower_components/**/*.js',
            './bower_components/**/*.css'
        ], {
            read: false // don't need to read, we only want paths
        }
    );

    return gulp.src('./*.html')
        .pipe(gulpinject(files_to_inject))
        .pipe(gulp.dest('./target'));
});
```

```
<html>
<head>
  <title>My index</title>
  <!-- inject:css -->
  <!-- endinject -->
</head>
<body>

  <!-- inject:js -->
  <!-- endinject -->
```

```
</body>
</html>
```

The inject:xxx comments define where the CSS and JS should go. Once you run the gulp task, the target folder should contain the same html file but with the appropriate tags added in.

Unlike wiredep, the gulp-inject plugin doesn't automatically order js and css files based on their dependencies. As such, you may need to manually order the files that you're injecting. See this stackoverflow post:
https://stackoverflow.com/questions/30841507/how-to-order-injected-files-using-gulp.

Injecting 3rd-party CSS/JS into HTML (bower+wiredep)

If you've downloaded your client-side dependencies through bower, you may want to automatically have those dependencies injected into your code. The most important reason for this is because the the dependencies are loaded in order -- meaning that if the dependencies you're using have a large dependency chain, then they'll all be properly loaded up.

To install…

```
~/myproject $ npm install --save-dev wiredep
~/myproject $ npm install -g bower
```

Usage example…

```
var gulp = require('gulp');
var wiredep = require('wiredep').stream; // must use stream for gulp

gulp.task('wiredep', function () {
    return gulp.src('./*.html')
        .pipe(wiredep({
            // options go here, if any
        }))
        .pipe(gulp.dest('./target'));
});
```

```
<html>
<head>
  <!-- bower:css -->
  <!-- endbower -->
</head>
<body>
```

```
  <!-- bower:js -->
  <!-- endbower -->
</body>
</html>
```

The bower:xxx comments define where the CSS and JS from bower should go. Once you run the gulp task, the target folder should contain the same html file but with the appropriate tags added in.

> **NOTE**: In certain cases, your source html file might get blanked out. If this happens, you'll end up getting an obscure error message… "TypeError: First argument must be a string, Buffer, ArrayBuffer, Array, or array-like object."

If you want to, you can tell bower to automatically run your gulp wiredep task after it installs all the components. You can do this by going to your .bowerrc file (not the one in /bower_components/, but the one in your main project directory) and adding in a postinstall script…

```
{
    "directory": "bower_components",
    "scripts": {
        "postinstall": "gulp wiredep"
    }
}
```

> **NOTE**: Like npm, if you download a package and want to install all the bower dependencies it declares, you do "bower install".

Combining JS/CSS files (gulp-useref)

This plugin helps combines multiple javascript files into a single javascript file + multiple CSS files into a single CSS file. It does this by looking at your HTML file, picking out the Javascript/CSS files, combining them, and then rewriting your HTML file to use the new single files.

> **NOTE**: This doesn't minify / compress the files. It just combines them.

To install...

```
~/myproject $ npm install --save-dev gulp-useref
```

Usage example…

```
var gulp = require('gulp');
var useref = require('gulp-useref');
```

```
gulp.task('default', function () {
    return gulp.src('app/*.html')
        .pipe(useref())
        .pipe(gulp.dest('dist'));
});
```

For this to work you need to explicitly define the region in which it operates in within the HTML. You do this using comments (just like with wiredep and gulp-inject)...

```
<html>
<head>
    <!-- build:css css/combined.css -->
    <link href="css/one.css" rel="stylesheet">
    <link href="css/two.css" rel="stylesheet">
    <!-- endbuild -->
</head>
<body>
    <!-- build:js scripts/combined.js -->
    <script type="text/javascript" src="scripts/one.js"></script>
    <script type="text/javascript" src="scripts/two.js"></script>
    <!-- endbuild -->
</body>
</html>
```

Comment format for defining the region is <!-- build:{js|css} <outputpath> --> … <!-- endbuild -->.

Minifying Images (gulp-imagemin)

To automatically compress JPEG/GIF/SVG/PNG files, you can use the gulp-imagemin plugin.

To install…

```
~/myproject $ npm install --save-dev gulp-imagemin
```

Usage example…

```
var gulp = require('gulp');
var imagemin = require('gulp-imagemin');

gulp.task(minify-images, () =>
    return gulp.src('src/images/*')
        .pipe(imagemin())
        .pipe(gulp.dest('dist/images'));
);
```

If you want to have special options used during the compression, you can pass in options. For example…

```
.pipe(imagemin([
    imagemin.gifsicle({interlaced: true}),
    imagemin.jpegtran({progressive: true}),
    imagemin.optipng({optimizationLevel: 5}),
    imagemin.svgo({
        plugins: [
            {removeViewBox: true},
            {cleanupIDs: false}
        ]
    })
]))
```

Minifying HTML (gulp-htmlmin)

To automatically minify your HTML files, you can use the gulp-htmlmin plugin.

To install…

```
~/myproject $ npm install --save-dev gulp-htmlmin
```

Usage example…

```
var gulp = require('gulp');
var htmlmin = require('gulp-htmlmin');

gulp.task('minify', function() {
  return gulp.src('src/*.html')
    .pipe(htmlmin({collapseWhitespace: true}))
    .pipe(gulp.dest('dist'));
});
```

Minifying CSS (gulp-csso)

To automatically minify/optimize your CSS files, you can use the gulp-csso plugin.

To install…

```
~/myproject $ npm install --save-dev gulp-csso
```

Usage example…

```
var gulp = require('gulp');
var csso = require('gulp-csso');

gulp.task('default', function () {
    return gulp.src('./main.css')
        .pipe(csso())
        .pipe(gulp.dest('./out'));
});

gulp.task('development', function () {
    return gulp.src('./main.css')
        .pipe(csso({
            restructure: false,
            sourceMap: true,
            debug: true
        }))
        .pipe(gulp.dest('./out'));
});
```

Minifying Javascript (gulp-uglify)

To automatically minify/optimize your JS files, you can use the gulp-csso plugin.

To install…

```
~/myproject $ npm install --save-dev gulp-uglify
```

Usage example…

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var pump = require('pump');

gulp.task('compress', function (cb) {
    return gulp.src('lib/*.js'),
        uglify(),
        gulp.dest('dist');
});
```

Other Tasks

Working on a Subset of Files in the Pipe (gulp-filter)

Sometimes you only want to work on a subset of the files that are passing through the pipe. You can do this using the gulp-filter plugin.

To install…

```
~/myproject $ npm install --save-dev gulp-gulpfilter
```

Usage example…

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var filter = require('gulp-filter');

gulp.task('default', () => {
    // Create filter instance inside task function
    const f = filter(['**', '!*src/vendor'], {restore: true});

    return gulp.src('src/**/*.js')
        // Filter a subset of the files
        .pipe(f)
        // Run them through a plugin
        .pipe(uglify())
        // Bring back the previously filtered out files (optional)
        .pipe(f.restore)
        .pipe(gulp.dest('dist'));
});
```

Giving File Revisions a Unique Name (gulp-rev/gulp-rev-replace)

You can make sure your files have unique names when they go to production by using the gulp-rev and gulp-rev-replace plugins. Essentially what this does is hash the file and add the hash to the end of the filename (gulp-rev), then go through any files that are referencing those files and update them to to point to the new filenames (gulp-rev-place).

This is useful for cache busting?

To install…

```
~/myproject $ npm install --save-dev gulp-rev gulp-rev-replace
```

Usage example…

```
var gulp = require("gulp");
var rev = require("gulp-rev");
var revReplace = require("gulp-rev-replace");

gulp.task("rev", ["dist:css", "dist:js"], function(){
  return gulp.src(["dist/**/*.css", "dist/**/*.js"])
    .pipe(rev())
    .pipe(gulp.dest(opt.distFolder))
    .pipe(rev.manifest())
    .pipe(gulp.dest(opt.distFolder))
})

gulp.task("revreplace", ["rev"], function(){
  var manifest = gulp.src("./" + opt.distFolder + "/rev-manifest.json");

  return gulp.src(opt.srcFolder + "/index.html")
    .pipe(revReplace({manifest: manifest}))
    .pipe(gulp.dest(opt.distFolder));
});
```

Running Script on File Change (gulp-nodemon)

If you want to run a command/script when something changes on the filesystem, the gulp-nodemon plugin is what you would use. The gulp-nodemon plugin uses nodemon to monitor a filesystem, and as soon as some file changes it will re-run some script.

In addition to that, you can run tasks and/or custom functions on start/restart/crash/exit.

To install…

```
~/myproject $ npm install --save-dev gulp-nodemon
```

Usage example…

```
var gulp = require('gulp');
var gulpnodemon = require('gulp-nodemon');

gulp.task('serve-dev', ['sometaskhere'], function () {
    var nodeOptions = {
        script: './src/server/app.js',
        delayTime: 1,
        env: {
            'PORT': 7203,
            'NODE_ENV': 'dev'
        },
        watch: ['./src/server']
    };

    return gulpnodemon(nodeOptions)
        .on('restart', [/* gulp tasks to run*/], function (ev) {
        /* func to run */
    })
        .on('start', [/* gulp tasks to run*/], function (ev) {
        /* func to run */
    })
        .on('crash', [/* gulp tasks to run*/], function (ev) {
        /* func to run */
    })
        .on('exit', [/* gulp tasks to run*/], function (ev) {
        /* func to run */
    });
});
```