Extra Notes - Grunt - Basic Usage

- Dr Nick Hayward

A brief introduction to using Grunt task runner.

Contents

- Intro
- Sample `grunt.initConfig`
- Install Grunt
- Create `Gruntfile.js`
- Example Grunt task
  - idempotent build tasks
  - consistency in Grunt config
  - globbing patterns
- Working with static assets
  - bundling static assets
  - Grunt and bundling
  - Grunt and asset minification
  - Grunt module bundler - Rollup
  - Grunt cleanup
- Working with image sprites
  - Grunt and image sprites
- Custom build task
- References

Intro Grunt is a build tool driven by specified configurations to ease setup of complex tasks for app development.

We may define *workflows* , which are then modified relative to development, testing, release, deployment &c. So, commonly we define *build tasks* for use with Grunt.

Such tasks and configuration is written in JavaScript, allowing developers to specify tasks for options such as

- compile
- minify
- testing
- spritesheets
- image compression
- module rollup
- …

Such build *tasks* perform an action in conjunction with *targets*, which help define a context.

For example, for a *minify* context we may specify targets for `templates` and `javascript` . In effect, relative to the each target, should the task produce a single bundle or simply minify each file individually. We may customise such end results to fit the needs of a given build.

These tasks will commonly be configured by passing an object to the `grunt.initConfig()` method. Then, we may describe the affected files, and pass any options to help modify required behaviour for a given task.

Tasks may be custom, and specific to a given project, or imported from available *plugins*. Such plugins are Node modules, describing one or more Grunt tasks.

Sample `grunt.initConfig()` An example Grunt config may be as follows

```
grunt.initConfig(
    {
        compile: {
            controllers: {
                files: ['./controllers/*.js']
            },
            services: {
                files: ['./services/*.js']
            }
        }
    }
);
```

Relative to the `initConfig()` method, we've passed the required object. This object includes initial child properties for each task, e.g. `compile` , and the required targets for the current task.

So, `compile` task with two initial targets for `controllers` and `services` .

Install Grunt To work with Grunt, we start by installing the required command line tool, `grunt-cli` .

```
npm install -g grunt-cli
```

The CLI tool is a global install, which then allows us to install, configure, and use Grunt specific to a given individual project.

For example, we might `cd` to a project's root directory and setup Grunt for local use. We need to start by defining the current project with a minimum Node setup, a basic `package.json` file for the project. A minimum config may include an empty object.

Then, we may install the Grunt tool as a dev dependency of the current project. For example,

```
npm install grunt --save-dev
```

After installation, the `package.json` will be updated to include metadata for the dev dependency install

```
{
    "devDependencies": {
        "grunt": "..."
    }
}
```

Create `Gruntfile.js` Grunt uses this file to load any required tasks, and configure them with any passed parameters.

This file, `Gruntfile.js` , should be saved in the root directory of the current project alongside the required `package.json` config.

This is the bare minimum for a `gruntfile.js` configuration

```
module.exports = function (grunt) {
    // register - default task alias
    grunt.registerTask('default', []);
}
```

This example shows the basic syntax and structure for Grunt config. For example, Grunt files are Node modules using CommonJS for their underlying structure.

`module` is a local object rather than a global object.

So, in this example, we're simply defining a task to run, in this example the *default*, when we run the `grunt` command.

The `default` task will be executed if no argument is passed at execution.

However, in this example, we're still only executing an empty task alias.

To execute tasks, we may define them in the array passed to the `registerTask()` method. e.g.

```
['lint', 'build']
```

In this example, the `lint` task will be executed first followed by the `build` task.

Example Grunt task Our first Grunt task will combine basic Grunt functionality with JSHint linting.

This linter is separate from the standalone CLI tool, and is a plugin module provided for use with Grunt tasks and a given build flow.

As with the main `grunt` package, we may install this plugin as a dev dependency. e.g.

```
npm install grunt-contrib-jshint --save-dev
```

We may then update the config for Grunt, the `Gruntfile.js`

```
module.exports = function(grunt) {
    grunt.initConfig(
        {
            jshint: ['Gruntfile.js']
        }
    );
    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.registerTask('default', ['jshint']);
};
```

In effect, we're updating Grunt to lint the config file, `Gruntfile.js` , and then load the JSHint plugin package. This will now setup the task for linting in the current project.

As `jshint` is the default task, we may execute it using the following command,

```
grunt
```

It will then return the output for this task's execution, e.g.

```
Running "jshint:0" (jshint) task
>> 1 file lint free.

Done.
```

idempotent build tasks A fundamental property of build tasks is *idempotency*.

i.e. repeated execution of a task should not produce different results…

For linting tasks, this might mean the same errors returned assuming no change in source code.

This property of idempotence, plus automated tasks, helps produce consistent, reliable results.

consistency in Grunt config A benefit of working with Grunt is the inherent consistency of config patterns for various defined tasks.

Configuration patterns for various tasks do not vary greatly, providing consistency in structure, syntax, and usage.

Config of the tasks themselves is consistent, whilst the execution and output may vary across workflows.

globbing patterns *Globbing* is a useful but simple concept for defining file path structures and usage.

In effect, globbing allows us to define includes and excludes for a given project structure. This means we do not need to keep a record of all files for various directories, associated assets, includes &c.

For example, if a developer adds a new script or stylesheet to a project directory, this will now be recorded and monitored as part of the various workflows.

We may also use this pattern to exclude various files and directories from a given build task.

A few patterns include the following samples

```
[
    'src/styles/*.css',
    'src/scripts/**/*.js',
    '!dev/tests/*.js'
]
```

The first example will simply match all files with an ending of `.css` in the defined directory. The second example follows a similar pattern, but it also abstracts the directory structure to include all child directories of the `src/scripts/` .

The last example follows the same underlying pattern as the first example. However, it is now *excluding* those matched files.

So, we may apply such globbing patterns to configs for Grunt &c.

Working with static assets When working with assets, we may consider various options to improve their usage in a project.

For example, we may initially consider bundling of static assets and minification of applicable files and resources.

bundling static assets Bundling is a simple concept, we're combining a project's required assets prior to release and deployment.

An inherent benefit is a reduction in network costs and transactions. The overall payload may increase, but a bundle commonly saves clients unnecessary network transactions and roundtrips to a server. So, we may reduce unnecessary network requests, general latency, TCP and TLS handshakes &c.

As a general concept, asset bundling literally appends each defined group of files to the end of the previous one. We might bundle all CSS files, or perhaps a library of JS files &c.

Fewer HTTP requests results in better performance.

Grunt and bundling We may use Grunt to easily configure various build targets to help bundle project assets. For example, we might combine required globbing patterns with a plugin package such as `grunt-contrib-concat` .

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

This plugin package helps concatenate defined files for improved asset management. e.g.

```
grunt.initConfig(
    {
        concat: {
            js: {
                files: {
                    'build/js/bundle.js': 'src/js/**/*.js'
                }
            }
        }
    }
);
```

So, we may now update our overall Grunt config to include concatenation

```
module.exports = function(grunt) {
    grunt.initConfig(
        {
            jshint: ['Gruntfile.js'],
            concat: {
                js: {
                    files: {
                        'build/js/bundle.js': 'src/js/**/*.js'
                    }
                }
            }
        }
    );

    // These plugins provide necessary tasks.
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-jshint');

  // Default task.
  grunt.registerTask('default', ['jshint', 'concat']);

};
```

and then call it as usual, using the `grunt` command at the terminal.

Grunt and asset minification Minification is another option for reducing file sizes, and strain on network requests.

In contrast to concatenation, minification commonly removes white space, shortens variable names, and optimises the underlying syntax tree for defined project code. The goal, of course, is to reduce the overall size of assets, thereby improving network usage and performance.

However, whilst minification reduces file sizes, it also makes the code practically unreadable for developers. Unfortunately, this is still not sufficient to hide any semblance of sensitive information. Minified code can be restored to a verbose output.

Another benefit of minification is that we may combine its usage with bundling options, thereby providing concatenation and minification for required projects.

For Grunt, we may use the plugin package `grunt-contrib-uglify` , which was originally meant to minify JavaScript files.

We may install this plugin package with the following command,

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

We need the `-es` plugin package to ensure support for ES6 (ES2015) code syntax and usage.

Then, we can update our initial Grunt config to include minification options

```
module.exports = function(grunt) {
    grunt.initConfig(
        {
            jshint: ['Gruntfile.js'],
            concat: {
                js: {
                    files: {
                        'build/js/bundle.js': 'src/js/**/*.js'
                    }
                }
            },
            uglify: {
                js: {
                    files: {
                        'build/js/mini.js': 'src/js/**/*.js'
                    }
                }
            }
        }
    );

    // These plugins provide necessary tasks.
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.loadNpmTasks('grunt-contrib-uglify-es');
  // Default task.
  grunt.registerTask('default', ['jshint', 'concat', 'uglify']);
};
```

We may also add minification for CSS, for example, using the `grunt-contrib-cssmin` plugin.

We need to install the Grunt module,

```
npm install grunt-contrib-cssmin --save-dev
```

and then add the following to include this package in the `Gruntfile.js` config,

```
grunt.loadNpmTasks('grunt-contrib-cssmin');
```

Then, we may update the build task for an example release distribution, e.g.

```
// build tasks with specific 'release' targets...
  grunt.registerTask('build:release', ['rollup:release', 'cssmin:release', 'uglify:release',
                     'buildMeta:release', 'clean']);
```

referencing the following task for `cssmin`

```
cssmin: {
    release: {
        options: {
        banner: '/* minified css file - basic-es-modules */'
        },
        files: {
      'build/css/mini.css': [
        'src/css/main.css',
        ]
        }
    }
```

```
},
```

Once the minified CSS stylesheet has been built, we can add a link to it in our `index.html` file

```
<!-- css styles - main -->
<link rel="stylesheet" href="./build/css/mini.css">
```

Grunt module bundler - Rollup We can also use a bundler specifically for handling ES Modules, *Rollup*.

For example, we might have a local library or collection of ES modules, which we need to bundle prior to minification.

To install Rollup, use the following command

```
npm install grunt-rollup --save-dev
```

Then, we may update the Grunt config to include module bundling.

```
...
rollup: {
    js: {
    options: {},
    files: {
        'temp/js/rolled.js': ['src/js/main.js'],
    },
    }
},
...
```

So, instead of using `concat` with modules, we may now use the specific `rollup` plugin package.

Combined with the `uglify` plugin, we have a single minified file for a library of modules.

Grunt cleanup One of the issues with build tools &c. is the potential for temporary files, directories &c. created at various stages of the build.

For example, in the above bundler task we create a `temp` directory for the bundled files. So, we now need to ensure that these are cleaned after we've created the minified file for distribution usage.

We can install the following Grunt plugin package

```
npm install grunt-contrib-clean --save-dev
```

and then update the Grunt config as follows,

```
...
clean: {
  folder: ['temp'],
}
...
```

So, our current `Grunt.initConfig` file is as follows

```
module.exports = function(grunt) {
    grunt.initConfig(
        {
            jshint: {
                all: ['src/**/*.js'],
                options: {
            'esversion': 6,
                    'globalstrict': true,
                    'devel': true,
                    'browser': true
            }
            },
            rollup: {
                js: {
                options: {},
                files: {
                'temp/js/rolled.js': ['src/js/main.js']
                },
                }
        },
            uglify: {
                js: {
                    files: {
                        'build/js/mini.js': 'temp/js/*.js'
                    },
                }
            },
            clean: {
            folder: ['temp']
            }
        }
    );

    // linting, module bundling, minification, directory cleanup...
  grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.loadNpmTasks('grunt-rollup');
    grunt.loadNpmTasks('grunt-contrib-uglify-es');
    grunt.loadNpmTasks('grunt-contrib-clean');

  // default task...
  grunt.registerTask('default', ['jshint', 'rollup', 'uglify', 'clean']);
};
```

Working with image sprites Another option for optimising development and assets is to work with *sprites*.

Image *sprites* use many images, allowing us to build a large file, which contains all of the defined images.

Then, instead of referencing each image file, we may now use the `background-position` , `width` , and `height` properties in CSS to locate a given part of the generate image sprite.

In effect, sprites are a useful way to bundle images. In client-side development, sprites are particularly useful for bundling icons and small images, which may be referenced in various pages and places multiple times.

Whilst we could maintain a manual copy of the sprite sheet and CSS for such development, it is easier and more efficient to use a build tool such as Grunt.

Grunt and image sprites We may install the following Grunt plugin package to work with Sprites,

```
npm install grunt-spritesmith --save-dev
```

The simple idea is to provide a directory of image files, and then combine them into a single spritesheet with accompanying CSS stylesheet for reference.

The generated CSS includes a class ruleset for each image, which may then be referenced in an app's HTML markup.

For example, we may add a sprite task to the current Grunt config

```
sprite: {
    icons: {
        src: 'src/assets/images/*',
        dest: 'build/img/icons.png',
        destCss: 'build/css/icons.css'
    }
},
```

This task will also convert multiple image file types, where applicable, to a single combined spritesheet. In the above `src` directory, `assets/images` , there are multiple image files including both JPG and PNG.

The CSS stylesheet is then autogenerated, and creates rulesets for each image. e.g.

```
.icon-egyptian-felucca-ride {
  background-image: url(../img/icons.png);
  background-position: 0px -768px;
  width: 550px;
  height: 412px;
}
.icon-giza-plateau {
  background-image: url(../img/icons.png);
  background-position: 0px 0px;
  width: 1024px;
  height: 768px;
}
```

To use the generated spritesheet, we load the CSS stylesheet in our app's HTML,

```
<!-- css styles - sprites -->
<link rel="stylesheet" type="text/css" href="./build/css/icons.css">
```

and then add the required CSS class to a HTML element, e.g.

```
<div class="icon-egyptian-felucca-ride"></div>
```

We may also add this generated CSS stylesheet to our earlier tasks to concatenate and uglify a project's CSS.

Custom build task There are many existing plugins available for Grunt.

However, we may find it necessary to reproduce a custom task for a particular project. Instead of repeating this task each time it is needed for a particular project, we may create a custom build task for Grunt.

For example, we might generate a file for a project, which includes build details for future reference. This might include developer name, version number, timestamp of builds, and so on. This file may then be referenced as the project develops.

```
// custom task
grunt.registerTask('buildMeta', function() {
    // define task options - incl. defaults
    const options = this.options({
        file: '.md',
        developer: "lead developer"
    });
    const timestamp = +new Date();
    const contents = `developer = ${options.developer}\ntimestamp = ${timestamp.toString()}`;
    // write details to file for project records...
    grunt.file.write(options.file, contents);
});
```

Then, we may modify the options by passing custom values to Grunt's `initConfig()` ,

```
buildMeta: {
    options: {
        file: 'build/meta.md',
        developer: 'spire & signpost'
    }
},
```

We may also add this to our `default` task

```
// default task...
grunt.registerTask('default', ['jshint', 'rollup', 'uglify', 'sprite', 'buildMeta', 'clean']);
```

or simply call it by name from the command line,

```
grunt buildMeta
```

References

- Grunt - JavaScript Task Runner