Grunt - Project Outline - Setup & Usage

- Dr Nick Hayward

A sample outline for using Grunt with project development, testing, &c.

Contents

Intro

We may consider Grunt, and other task runners and build tools, relative to build distributions and development environments.

Initial setup and install

For a new project, we may begin by initialising a *Git* repository in the root directory.

We'll also add a `.gitignore` file to our local repository. This allows us to define files and directories not monitored by Git's version control.

Then, we initialise a new NodeJS based project using *NPM*. Defaults for `npm init` are initially fine for a standard project,

```
npm init
```

As the project develops and matures, we may modify accordingly this initial metadata.

Basic project layout may follow a sample directory structure,

```
.
|-- build
|   |-- css
|   |-- img
|   |-- js
|-- src
|   |-- assets
|   |-- css
|   |-- js
|   |__ app.js
|-- temp
|-- testing
|__ index.html //applicable for client-side, webview apps &c.
```

This sample needs to be modified relative to a given project, and `build`, `temp`, and `testing` will include files and generated content from various build tasks.

The `build` and `temp` directories will be created and cleaned automatically as part of the build tasks. They do not need to be created as part of the initial directory structure.

This example structure adds an `index.html` file to the root of the project structure for client-side and webview based development, and then references the appropriate directory for each environment.

This structure includes `build` directories, which we may not add until build tasks for a `release` distribution. These commonly include bundling, minification, uglifying, &c. Again, the `build` directory will be part of a build task.

We may also update our project's `.gitignore` file,

```
.DS_Store
node_modules/
*.log
build/
temp/
```

install and configure Grunt

We can start by installing and configuring Grunt for the above sample project structure

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

This install assumes a global scope for the NPM package `grunt-cli`, and saves the metadata to `package.json` for development builds only.

Grunt config

To use Grunt with a project, we add a config file, `Gruntfile.js`, which includes initial exports for tasks and targets.

Then, we may load and register the required tasks.

`Gruntfile.js` **- initial exports**

Grunt config is again dependent on the specifics of a particular project.

However, we may add some common options for linting, build distributions, minification and bundling, uglifying, sprites &c.

```
module.exports = function(grunt) {
        grunt.initConfig(
                {
                        jshint: {
                                all: ['src/**/*.js'],
                                options: {
                        'esversion': 6,
                                        'globalstrict': true,
                                        'devel': true,
                                        'browser': true
                }
                        },
                        rollup: {
                                release: {
                        options: {},
                        files: {
                        'temp/js/rolled.js': ['src/js/main.js'],
                        },
                                }
                },
                        uglify: {
                                release: {
                                        files: {
                                                'build/js/mini.js':
'temp/js/*.js'
                                        },
                                }
                        },
                        sprite: {
                                release: {
                                        src: 'src/assets/images/*',
                                        dest: 'build/img/icons.png',
                                        destCss: 'build/css/icons.css'
                                }
                        },
```

```
                        clean: {
                        folder: ['temp'],
                        }
                }
        );

};
```

Use of `rollup` will depend upon required support for modules, including ES modules within JavaScript apps.

Then, we may add custom tasks such as metadata generation,

```
buildMeta: {
        options: {
                file: './meta.md',
                developer: 'debug tester',
                build: 'debug'
        }
},
```

We may add tasks for CSS &c. as we continue to develop the project.

`Gruntfile.js` - use tasks

After defining the exports for tasks and targets, we can load the required Grunt plugin modules, and register the required tasks.

We may run these registered tasks together or separately relative to distribution and environment.

For example, we can load the required plugins for the above tasks,

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

These correspond to the plugin packages installed using NPM for the current project.

```
npm install grunt-contrib-jshint --save-dev
npm install grunt-rollup --save-dev
npm install grunt-contrib-uglify-es --save-dev
npm install grunt-spritesmith --save-dev
npm install grunt-contrib-clean --save-dev
```

`Gruntfile.js` **- register custom task**

We can then register a custom task for various targets in the builds.

For example,

```
// custom task - build meta for default debug
grunt.registerTask('buildMeta', function() {
        console.log('debug build...');
        const options = this.options();
        metaBuilder(options);
});

//custom task - build meta for release
grunt.registerTask('buildMeta:release', function() {
                console.log('release build...');
        // define task options - incl. defaults
        const options = this.options({
                file: 'build/release_meta.md',
                developer: "spire & signpost",
                build: "release"
        });
        metaBuilder(options);
});
```

`Gruntfile.js` **- register builds**

We can then register some build tasks, which combine the options from the config. This provides the execution of staggered tasks for a single build call.

For example, a debug build may include linting, custom metadata, and a clean task

```
// debug build tasks - default tasks during development...
grunt.registerTask('build:debug', ['jshint', 'buildMeta', 'clean']);
```

This may be followed by a build process for staging or release,

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

Then, we may run and test Grunt for the current project,

```
grunt build:debug
```

or

```
grunt build:release
```

relative to project requirements.

Development with environments

As we develop more complex apps, we need to consider how we configure and use such build tools with various environments.

- development
- staging
- production / release

We can fit either a *debug* or *release* distribution build into each of these environments.

environment setup - development

App development will primarily focus on a `debug` distribution to provide tasks such as linting, testing, metadata, watch, &c. This will become the common distribution for active, ongoing development.

We also need to ensure that environment variables are aggregated to allow the app to run. These may be stored in the same manner regardless of `debug` or `release`. The difference is the use of encryption, and the nature of the required environment configs.

Bundling with minification and uglifying will usually be added to a project as part of the `release` distribution. It serves little practical benefit for ongoing active development of a project's code base.

So, we may define a common structure for Node based apps as follows

```
.
|-- debug
|-- src
|   |-- assets
|   |-- js
|-- temp
|-- testing
|__ app.js
```

We can develop the app, including the app source code, in the `src` directory. Then, we may build our app in the `debug` directory each time we need to check and debug usage.

Any temporary build artifacts may be added to the `temp` directory, and then cleaned after each build workflow has been completed. In effect, each time we complete a call to `build:debug`, we may clean, where applicable, the build artifacts. We may also choose to combine `debug` and `temp` into a single `temp` directory, depending upon project requirements.

For a client-side or mobile hybrid app, we may slightly modify this directory structure as follows

```
.
|-- debug
|   |-- css
|   |-- img
|   |-- js
|-- src
|   |-- assets
|   |-- css
|   |-- js
|   |__ app.js
|-- temp
|-- testing
|__ index.html
```

The `assets` directory may include raw image files, icons, &c. For the `debug` distribution, we may test builidng these image assets as sprites, which are all added to the `img` directory during the build.

We may also use *image optimisation* at this stage, in particular to test UI and UX performance.

Part of the `debug` distribution is the use of `watch` for live reloading, and `nodemon` for Node.js based apps.

We might also consider tasks to aggregate logging within the app's code. This may include explicit `console.log()` statements, and error handling.

For a release distribution, for example, we might add a custom task to remove all development logging, such as `console.log()` statements, from the project's JS code.

environment setup - development Grunt config

We can now update our Grunt config to use a `debug` distribution in the current *development* environment.

For example, we need to add any required build options for debug, and then integrate the required environment config variables &c.

We can start with *unencrypted JSON* files, which may contain defaults for options such as the current environment, and the server's port number.

A sample JSON object is as follows,

```
{
        "NODE_ENV": "development",
        "PORT": 3826
}
```

Then, we may define some additional project directories for encrypted and decrypted config files.

```
.
|-- env
|   |-- defaults
|   |-- private
|   |-- secure
```

So, `env/defaults` contains the unencrypted defaults, as defined in `defaults.json`, whilst `env/private` includes decrypted secure files. `env/secure` should be reserved for encrypted files, which we may add to version control. However, `env/private` should not be commited to version control.

There are a few different options for file encryption, including RSA based public/private keys, and GNU Privacy Guard (GPG, or GnuPG).

encrypt with RSA public key

For distributed files with encryption, we may use RSA tools to ensure a file is encrypted prior to sharing. RSA usage is as follows,

- create private key (**do not share**)
- use key to encrypt sensitive files
- use encrypted file with code base
    - change unencrypted file and then encrypt (re-run as needed...)
- encrypted file can only be accessed with public key

On OS X, for example, we may encrypt a file using `ssh-keygen`. A good intro is available at the following URL,

- https://www.ssh.com/ssh/keygen/

However, if we want to create a standard public/private key pair, we may use the following initial terminal command

```
ssh-keygen -t rsa -b 4096
```

This will create a public `id_rsa.pub` key with a matching `id_rsa` private key in a hidden directory, `.ssh`, in the user's home directory,

```
cd ~/.ssh
```

However, we may also create these keys with custom names for a given project,

```
ssh-keygen -f ~/.ssh/basic-env-node -t rsa -b 4096
```

A common use of such public/private key pairs is for remote server access using `ssh`. However, we may also use such keys to encrypt files.

To encrypt larger files, including JSON, txt &c., we need to use a public key in `.pem` format. We need to generate a `.pem` version of our current public key,

However, we start by converting the private key,

```
openssl rsa -in id_rsa -outform pem > id_rsa.pem
```

and then the matching public key

```
openssl rsa -in id_rsa -pubout -outform pem > id_rsa.pub.pem
```

Each key is defined relative to its current location, e.g. `~/.ssh/id_rsa` is common for OS X, Linux &c.

It is now safe to share the generated `id_rsa.pub.pem` key file.

Then, we need to generate a 256bit (32 byte) random key for encrypting the file,

```
openssl rand -base64 32 > key.bin
```

and then we encrypt the key itself,

```
openssl rsautl -encrypt -inkey id_rsa.pub.pem -pubin -in key.bin -out
key.bin.enc
```

Finally, we may now encrypt the large file

```
openssl enc -aes-256-cbc -salt -in defaults.json -out defaults.json.enc -
pass file:./key.bin
```

We can now send the `.enc` files to another developer or add them to version control. To decrypt the encoded file,

```
openssl rsautl -decrypt -inkey id_rsa.pem -in key.bin.enc -out key.bin

openssl enc -d -aes-256-cbc -in defaults.json.enc -out defaults.json -pass
file:./key.bin
```

We will now have the decryted file `defaults.json` with the required settings for the local system.

n.b. we should also verify the *hash* of the file with the recipient or even sign it with a private key. This is to avoid a *man-in-the-middle* attack.

sign and verify a file

We can add an extra layer of verification for a file by adding a generated signature for a given RSA key pair.

As above, we may again use `openssl` to sign and verify a required signature.

To sign a file, we may use the digest function, `dgst`, provided by `openssl`. Documentation for `dgst` may be found at the following URL,

- openssl - dgst

For example, to sign a file using SHA-256 with binary file output

```
openssl dgst -sha256 -sign id_rsa.pem -out signature.sign defaults.json
```

In this example, we are creating a signature file for the file, `defaults.json`, using the local private key `id_rsa.pem`.

We may then check and verify this signature for the file `defaults.json`,

```
openssl dgst -sha256 -verify id_rsa.pub.pem -signature signature.sign
defaults.json
```

This check requires the shared public key for the file `defaults.json`.

So, we may now sign the encrypted file, and provide the signature for verification purposes,

```
openssl dgst -sha256 -sign id_rsa.pem -out signature.enc.sign
defaults.json.enc
```

environment setup - hosted solutions

For *staging* and *production*, we need to consider hosted solutions.

We may choose to host these environments on custom servers or, commonly, we may use a cloud service such as Heroku.

Heroku provides a command-line interface for managing environment variables per project.

For example, the Heroku CLI tool may be used to manage hosted projects from a terminal.

This CLI tool allows a developer to easily set environment variables for a project. We can define the environment as *staging*, for example, and customise variables specific to this hosting requirement.

```
heroku config:set NODE_ENV=staging
```

We might also modify the port number for a particular server or, perhaps, update the execution mode to debug or release.

Merging config sources

As a project develops, we may produce various sources of configuration.

As noted above, this may include sources such as JSON files, JavaScript objects, environment variables, process arguments, and so on.

To help merge such disparate config sources, we may add an NPM module such as `nconf`

- nconf

or we may simply load environment variables from a project's `.env` file using the package `dotenv`

- dotenv

sample waterfall with nconf

With `nconf`, we may bundle various config stages for a project.

For example,

```
const nconf = require('nconf');
nconf.argv();
nconf.env();
nconf.file('dev', 'development.json');
module.exports = nconf.get.bind(nconf);
```

In effect, we're grabbimg config variables and settings from the defined stores in a defined cascading order.

The order is prioritised, allowing overrides and defaults at various stages of the cascade. In the above example, if a value is given in the command arguments, `argv`, this will take precedent over subsequent stages.

Continuous development

Continuous development allows a developer to work on app code &c. without many of the customary interruptions, such as server reboots, code refreshes, debugging, linting &c.

Continuous development often reduces the repetitive tasks in a development flow, thereby automating processes and development.

A build process may be automated and run whenever a pertinent change is detected.

add a `watch`

We may add a ***watch*** task to a build flow to allow a rebuild each time a given file is edited and then saved.

For Grunt, we may add the plugin module `grunt-contrib-watch`.

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

and update the Grunt config,

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

This plugin watches the file system for code changes in a tracked project, and runs the affected tasks.

A basic `watch` example might include the following

```
watch: {
        js: {
                tasks: ['jshint:client'],
                files: ['src/**/*.js']
        }
}
```

This continuously checks the `src` directory for a JavaScript file change or addition, which will then run the `jshint:client` task. This type of `watch` provides a broad approach to managing project changes.

We may then include additional ***targets*** relative to project requirements. Similar to the above JS pattern, we may add further JS specific targets, CSS, sprites &c.

As expected, we may also define a separate build tasks to use `watch`, e.g.

```
// dev tasks - combine debug with watch
grunt.registerTask('dev', ['build:debug', 'watch']);
```

which we may call as follows,

```
grunt dev
```

This will first execute the tasks for `build:debug`, and then start ***watching*** the specified targets.

live reload

We may also use `watch` to add support for ***live reloads***. There is built-in support with the `grunt-contrib-watch` plugin.

The reload option uses ***web sockets***, a technology originally designed for browser based real-time communication and synchronisation. The LiveReload option listens for changes to monitored files, directories &c. It may then reload and refresh the current active app.

Support for the LiveReload task may added as follows,

```
livereload: {
        options: {
                livereload: true
        },
        files: ['build/**/*', './*.html'],
},
```

This will provide a live reload server, which usually runs at `localhost:35729`. This object includes a property to confirm `livereload`, and then defines the files to watch to initiate a reload. In this example, we're watching the `build` directory, and its children, and then the root directory for any HTML files. This will include, of course, any changes to the default `index.html` file.

However, this server does not actually reload the app for us. We need to use a server to host the app, which is then monitoring this `livereload` server.

To help with this monitoring setup, `livereload` also provides a setup script for the test app. We may either add a link to this script in our project's `index.html` file,

```
<script src="http://localhost:35729/livereload.js"></script>
```

or use a Grunt plugin, `grunt-contrib-connect` to automatically inject it in our app's code. This is the preferred option for ongoing development.

We may install this plugin as follows,

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

and then update the `Gruntfile.js` config as follows,

```
connect: {
        server: {
                options: {
                  port: 8080,
                  base: '.',
                  hostname: '*',
                  protocol: 'http',
                  livereload: true,
                }
        },
},
```

To use these plugins, we need to update the required build tasks, e.g. we'll add connect and livereload support to the `dev` build task

```
// dev tasks - combine debug with watch, live server, and live reload
grunt.registerTask('dev', ['build:debug', 'connect', 'watch']);
```

We may then run this build task,

```
grunt dev -v
```

The `-v` flag outputs verbose messages to help us initialy check everything is running as expected.

monitor `node`

We may also add a monitor for Node using the package `nodemon`.

Usage will depend on app requirements, and whether we are integrating any running Node processes, for example *Express*.

`nodemon` install is as follows,

```
npm install nodemon
```

or add the `-g` flag for system-wide install.

We may then use `nodemon` to run Node based apps instead of the standard `node` command.

Add CSS support

App styles will, customarily, include a combination of CSS stylesheets and dynamic JavaScript based style properties.

To work with CSS stylesheets, similar to JavaScript files, we may consider a Grunt task for minifying these files.

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 a release distribution

```
// 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">
```

We may then update our `watch` task by adding the following for CSS,

```
css: {
        files: ['src/**/*.css'],
        tasks: ['cssmin:release']
},
```

We may then run our usual Grunt build tasks to minify the CSS stylesheets, and watch for any updates and changes.

Watch update

The current `watch` task includes support for CSS, JS, and HTML.

It includes checks for modifications to any of the defined `src` directories for CSS and JS, plus any HTML files in the app's root directory.

A working `watch` task is as follows,

```
watch: {
        js: {
                files: ['src/**/*.js'],
                tasks: ['jshint:client', 'rollup:release',
'uglify:release']
        },
        css: {
                files: ['src/**/*.css'],
                tasks: ['cssmin:release']
        },
        html: {
                files: ['./*.html']
        },
        livereload: {
                options: {
                        livereload: true
                },
                files: ['build/**/*', './*.html'],
        },
},
```