Improving Your Workflow with GRUNT

JavaScript Web Development

Author: Tim Clark

Update Date: 06/14/2014
Version: 1.0

Document Conventions

`command/task`	Indicates an action to be performed by the reader
`output/results`	Indicates the expected result of an action

Technologies Required

Note: Take the time to read about each of these technologies prior to completing the tutorial.

NODE.js + NPM

JavaScript for the Server Side (Use: Chrome JS Engine runs Standalone on Server)
http://nodejs.org/

GRUNT

The JavaScript Taskrunner (Use: Repetitive Task Automation)
http://gruntjs.com/

BOOTSTRAP

Front-end Template (Use: getting every project up and running faster)
http://getbootstrap.com

JSHINT

JavaScript Code Quality Analyzer (Use: Writing Better JS Code)
http://www.jshint.com/

JASMINE

Behavior Driven JavaScript (Use: Unit Testing All Custom JavaScript)
http://jasmine.github.io/2.0/introduction.html

UGLIFYJS

JavaScript Code Compression (Use: Parse/Compress/Beautify JS Code)
https://github.com/mishoo/UglifyJS

SMUSH.IT

Yahoo! Image Compression Service (Use: Compressing Images for Production Deployment) http://www.smushit.com/ysmush.it/

Getting Started

Node.js

First we need to install the runtime on which our workflow will be built, to accomplish this first install Node.js by visiting its website and following the instructions for your specific operating system. Once installed verify the installation was successful by executing the following command in a command prompt (Windows) or terminal (Linux/Mac), we should get back the currently installed node version.

```
node -v
v0.10.28
```

GRUNT

Next we need a system to automate all of the repetitive tasks to must perform as we develop. This is accomplished through the GRUNT automation system. GRUNT is installed using Node's package manager called NPM, it allows us to download and install third party libraries for use within Node. This is very similar to DLL's in .Net and Packages in Java, the only difference is the fact that all Node packages are registered with the NPM service and all we simply have to do to get a package is call for it from the service.

Install GRUNT by running the following command and verifying the install was successful, we should see similar output.

```
npm install -g grunt-cli
npm http GET https://registry.npmjs.org/grunt-cli
…
…
/usr/local/bin/grunt -> /usr/local/lib/node_modules/grunt-cli/bin/grunt
grunt-cli@0.1.13 /usr/local/lib/node_modules/grunt-cli
├── resolve@0.3.1
├── nopt@1.0.10 (abbrev@1.0.5)
└── findup-sync@0.1.3 (lodash@2.4.1, glob@3.2.11)
```

Bootstrap + Template

We can now create the skeleton of our application. We will be using a common HTML5 front-end template package called Bootstrap for this purpose. Download the pre-packaged Bootstrap archive from the LMS and unzip it. We now have the skeleton of our application.

Configuration

Node Package File

With the underlying workflow systems in place, it is now time to start wiring things together. To start we need to tell Node about our application as well as have a place to list all of the Node packages we are using in our workflow. This is done by creating a Node package file. Node package files are simply plain text files with the file extension .json and contain JSON formatted text following the NPM specification located at https://www.npmjs.org/doc/json.html.

In our new application's root directory, create a file with the name **package.json**. Place the following JSON as the contents of that file.

```
{
  "title": "My Web Application",
  "name": "my-web-app",
  "version": "0.1.0"
}
```

The **title** gives our Node package a human-friendly name while the **name** is meant for the machine to identify it. Of course **version** simply allows us to specify what version of the application we are currently working on.

GRUNT

Next we need to specify all of the dependencies (additional Node packages) we are using for our workflow. This can be done by entering each dependency into the package.json file manually or we can let NPM do it for us at the command line when we install each package using the command format: npm install <package-name> --save-dev. Lets add GRUNT as a development dependency now using the following command.

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

JSHINT

The GRUNT specific JSHINT Node package, when added as a dependency to our application, allows us to perform automated code analysis on our project, checking for JavaScript errors as well as enforcing common coding conventions so that our JS code is more maintainable and readable.

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

Jasmine

The GRUNT specific Jasmine Node package, when added as a dependency to our application, allows us to perform automated unit testing on all of our custom JavaScript.

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

Uglify

The GRUNT specific Uglify Node package, when added as a dependency to our application, allows us to perform automated JavaScript compression for production environment. Compressing JS saves on download time and bandwidth and thus improves web application performance.

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

Smushit

The GRUNT specific Smushit Node package, when added as a dependency to our application, allows us to compress all bitmap images within our project for the production environment. Just like with compressed JS or CSS, images with optimized compression save on download time and bandwidth making our application faster.

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

Complete Package.json

In the end our package.json should now look like this:

```
{
  "title": "My Web Application",
  "name": "my-web-app",
  "version": "0.1.0",
  "devDependencies": {
    "grunt": "^0.4.5",
    "grunt-contrib-jasmine": "^0.6.5",
    "grunt-contrib-jshint": "^0.10.0",
    "grunt-contrib-uglify": "^0.5.0",
    "grunt-smushit": "^1.3.0"
  }
}
```

GRUNT Configuration File

This config file is used to store settings and configuration information specific to GRUNT and all of the GRUNT specific Node packages we have as dependencies. It is also where all of the automated tasks we wish GRUNT to perform for us are located. To begin we will need to create a file with the name **GruntFile.js**. Add the following 'wrapper' function to the file, this is where all GRUNT configuration information will be stored.

```
module.exports = function(grunt){
    // Configuration here
    // Plugins here
    // Tasks here
};
```

GRUNT Plugin Loading

Before GRUNT can use any of the plugins we have added (jshint, jasmine, etc.), we have to first register them with GRUNT. Add the following lines to the 'plugin' section of the GruntFile.js we just created.

```
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-jasmine');
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-smushit');
```

GRUNT Plugin Task Registration

Now that GRUNT is aware of the plugins, we now need to register each plugin as having tasks associated with them. We do this by adding the following lines to the 'tasks' section of the GruntFile.js.

```
grunt.registerTask('default', ['jshint', 'jasmine', 'uglify', 'smushit']);
```

GRUNT Configuration Initialization

It is now time to configure all settings and options for GRUNT and all of the plugins. This is done by adding the following to our GruntFile.js.

Initializing the Configuration

GRUNT must of course be initialized and configured. This is done by making a call to the **grunt.initConfig** function, passing it an object literal containing configuration attributes.

```
grunt.initConfig({
    pkg: grunt.file.readJSON('package.json')
});
```

JSHINT Config

Since JSHINT evaluates and analyzes all JavaScript files in the project, it needs to be told where within the project the JS files live. This is done by adding a **jshint** specific attribute with an object literal holding all configuration attributes.

```
,
    jshint: {
        options: {
            strict: false
        },
        all: ['GruntFile.js', 'js/<%= pkg.name %>.js']
    }
```

Jasmine Config

For Jasmine to execute all unit tests within the project, it needs to know the location of all test files. This is done by adding a **jasmine** specific attribute with an object literal holding all configuration attributes.

```
,
    jasmine: {
        pivotal: {
            src: 'js/*.js',
            options: {
                specs: 'spec/*Spec.js',
                helpers: 'spec/*Helper.js'
            }
      }
    }
```

Uglify Config

In order for Uglify to compress all JavaScript files within the project, it needs to know the location of all JS files. This is done by adding an **uglify** specific attribute with an object literal holding all configuration attributes.

```
,
    uglify: {
        all: {
            files: {
                'js/<%= pkg.name %>.min.js': 'js/<%= pkg.name %>.js'
            }
        }
    }
```

Smushit Config

In order for Smushit to compress all image files within the project, it needs to know the location of all image files. This is done by adding an **smushit** specific attribute with an object literal holding all configuration attributes.

```
,
    smushit: {
        path: {
            src: 'img/'
        }
    }
```

Completed GRUNT Config File

In the end our GRUNT config file should look just like this:

```
grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    jshint: {
        options: {
            strict: false
        },
        all: ['GruntFile.js', 'js/<%= pkg.name %>.js']
    },
    jasmine: {
        pivotal: {
            src: 'js/*.js',
            options: {
                specs: 'spec/*Spec.js',
                helpers: 'spec/*Helper.js'
            }
      }
    },
    uglify: {
        all: {
            files: {
                'js/<%= pkg.name %>.min.js': 'js/<%= pkg.name %>.js'
            }
        }
    },
    smushit: {
        path: {
            src: 'img/'
        }
    }
});
```

Running GRUNT Tasks

Now that all of the GRUNt tasks have been set up, it is time to test the automation of those tasks and see if our new workflow works as expected.

With a terminal or command prompt open at the root folder of the application, all we need to do is execute GRUNT by running the GRUNT command.

```
grunt
```

Upon execution of the command we should see JSHINT run first and pass with 1 file lint free (the only JS file in our project, provided by the Boilerplate template). GRUNT will then execute with the following warning. This is due to the fact that Jasmine is attempting to run all unit tests in the project but cannot find a single test file (because we have not yet created any tests).

```
Aborted due to warnings
```

To make sure that all of our other GRUNT plugins are functioning properly, we can use the --force flag on the GRUNT command to force GRUNT to run all tasks even if one fails or creates a warning.

```
grunt --force
```

The output should now show results for all of our plugins, now including uglify and smushit, and a message stating all GRUNT tasks were successfully run.

```
Done, without errors.
```

And.. We’re Done!