JavaScript Application Design: Picking your build tool

By Nico Bevacqua, author of *JavaScript Application Design*

In this overview, I will help you understand the differences between the three build tools I use most often in front-end development workflows: grunt, npm, and Gulp.

Deciding on a technology is always difficult. You don't want to make commitments you can't back out of, but eventually you have to choose something. Committing to a build technology is no different in this regard: it's an important choice and you should treat it as such.

For the purposes of my book, I decided on Grunt as my build tool of choice. I made an effort to not go overboard on Grunt-specific concepts, but rather to explain build processes in the grand scheme of things, using Grunt as an accessory—the means to an end. I chose Grunt for several reasons; a few of these are shown in the following list:

- Grunt has a healthy community around it, even on Windows.
- It's widely popular; it's even used beyond the Node community.
- It's easy to learn; you pick plugins and configure them. No advanced concepts are used and no prior knowledge is needed.

These are all good reasons to use Grunt to teach build processes in a book, but I want to make it clear I don't think Grunt is the single best option out there; other popular build tools might fit your needs better than Grunt.

I wrote this overview to help you understand the differences between the three build tools I use most often in front-end development workflows:

- Grunt, the configuration-driven build tool
- npm, a package manager that can also double as a build tool
- Gulp, a code-driven build tool that's somewhere between Grunt and npm

I'll also lay out the situations in which a particular tool may be better than the others.

For the purposes of this overview, I will assume that you have basic knowledge of Grunt. As a first step, let's discuss where Grunt excels.

Grunt: the good parts

The single best aspect of Grunt is its ease of use. It enables programmers to develop build flows using JavaScript almost effortlessly. All that's required is searching for the appropriate plugin, reading its documentation, and then installing and configuring it. This ease of use means members of large development teams, who are often of varying skill levels, don't have any trouble tweaking the build flow to meet the latest needs of the project. The team doesn't need to be fluent in Node, either; they need to add properties to the configuration object and task names to the different arrays that make up the build flow.

Grunt's plugin base is large enough that you'll rarely find yourself developing your own build tasks, which also enables you and your team to rapidly develop a build pro- cess. This rapid development is crucial if you're going for a build first approach, even when taking small steps and progressively developing your build flows.

It's also feasible to manage deployments through Grunt, as many packages exist to accommodate for those tasks, such as `grunt-git`, `grunt-rsync`, and `grunt-ec2`.

Grunt: the bad parts

Where does Grunt fall short? It may get too verbose if you have a significantly large build flow. It's often hard to make sense of the build flow as a whole once it has been in development for a while. When the task count in your build flows gets to the double digits, it's almost guaranteed that you'll find yourself having to run targets that belong to the same task individually, so you can compose the flow in the right order.

Because tasks are configured declaratively, you'll also have a hard time figuring out the order in which tasks get executed. In addition, your team should be dedicated to writing maintainable code when it comes to your builds. In the case of Grunt, you'll maintain separate files for the configuration of each task, or at least for each of the build flows that your team uses.

Now that we've identified the good and the bad in Grunt, as well as the situations in which it might be a good fit for your project, let's talk about npm: how it can be used as a build tool and its differences from Grunt.

npm as a build tool

To use npm as a build tool, you'll need a package.json file and npm itself. Defining tasks for npm is as easy as adding properties to a `scripts` object in your package man- ifest. The name of the property will be used as the task name, and the value will be the command you want to execute. The following snippet represents a typical pack- age.json file, using the JSHint command-line interface to run a linter through your JavaScript files and check for errors. Using npm, you can run any shell command at your disposal:

Grunt in a nutshell

Grunt has the following benefits:

- Thousands of plugins that do what you need.
- Easy-to-understand and tweak configuration.
- Only a basic understanding of JavaScript is necessary.
- Supports cross-platform development. Yes, even Windows!
- Works great for most teams.

Grunt has a few drawbacks:

- Configuration-based build definitions become increasingly unwieldy as they grow larger.
- It's hard to follow build flows when there are many multitarget task definitions involved.
- Grunt is considerably slower than other build tools.

```
{
  "scripts": {
    "test": "jshint . --exclude node_modules"
  },
  "devDependencies": { "jshint": "^2.5.1"
  }
}
```

Once the task is defined, it can be executed in your command line by running the fol- lowing command:

```
npm run test
```

Note that npm provides shortcuts for specific task names. In the case of `test`, you can do `npm test` and omit the `run` verb. You can compose build flows by chaining `npm run` commands together in your script declarations. The following listing allows you to run the `unit` task right after the `lint` task by executing the `npm test` command.

Listing 1 Chaining npm `run` commands together to make build flows

```
{
  "scripts": {
    "lint": "jshint . --exclude node_modules",
    "unit": "tape test/*", "test":
    "npm run lint && npm run unit"
  },
    "devDependencies": {
    "jshint": "^2.5.1",
    "tape": "^2.10.2"
  }
}
```

You can also schedule tasks as background jobs, making them asynchronous. Suppose you have the following package file, where you'll copy a directory in your JavaScriptbuild flow and

compile a Stylus style sheet during your CSS build flow (Stylus is a CSS preprocessor). In this case, running the tasks asynchronously is ideal. You can achieve that using `&` as a separator, or after a command, as shown in the following listing of your package manifest. Afterward, you can execute `npm run build` to process both steps concurrently.

Listing 2 Using Stylus

```
{
  "scripts": {
    "build-js": "cp -r src/js/vendor bin/js",
    "build-css": "stylus src/css/all.styl -o bin/css",
      "build": "npm run build-js & npm run build-css"
  },
  "devDependencies": { "stylus": "^0.45.0"
  }
}
```

Sometimes a shell command won't suffice, and you may need a Node package such as `stylus` or `jshint`, as you saw in the last few examples. These dependencies should be installed through npm.

Installing npm task dependencies

The JSHint CLI isn't necessarily available in your system, and you have two ways to install it:

- Globally, when using it from your command line
- Adding it as a devDependency, when using it in an `npm run` task

If you want to use the tool directly from your command line, and not in an `npm run` task, you should install it globally using the `-g` flag in the following command:

```
npm install -g jshint
```

If you're using the package in an `npm run` task, then you should add it as a `dev-Dependency`, as shown in the following command. That allows npm to find the JSHint package on any system where the package dependencies are installed, rather than expecting the environment to have JSHint installed globally. This applies to any CLI tools that aren't readily available in operating systems.

```
npm install --save-dev jshint
```

You aren't limited to using only CLI tools. In fact, npm can run any shell script. Let's dig into that!

Using shell scripts in npm tasks

The following example is a script that runs on Node and displays a random emoji string. The first line tells the environment that the script is in Node.

```
#!/usr/bin/env node
  var emoji = require('emoji-random');
  var emo = emoji.random();
  console.log(emo);
```

If you place that script in a file named `emoji` at the root of your project, you'd have to declare `emoji-random` as a dependency and add the command to the `scripts` object in the package manifest:

```
{
  "scripts": { "emoji": "./emoji"
  },
  "devDependencies": { "emoji-random": "^0.1.2"
  }
}
```

Once that's out of the way, running the command is merely a matter of invoking `npm run emoji` in your terminal, which will execute the command you specified as the value for `emoji` in the `scripts` property of your package manifest.

npm and Grunt compared: the good and the bad

Using npm as a build tool has several advantages over Grunt:

- You aren't constrained to Grunt plugins, and you can take advantage of all of npm, which hosts tens of thousands of packages.
- You won't need any additional CLI tooling or files other than `npm`, which you're already using to manage dependencies and your `package.json` manifest, where dependencies and your build commands are listed. Because `npm` runs CLI tools and Bash commands directly, it'll perform way better than Grunt could.

Take into account that one of the biggest disadvantages of Grunt is the fact that it's I/O bound. Most Grunt tasks read from disk and then write to disk. If you have several tasks working on the same files, chances are that the file will be read from disk multi- ple times. In Bash, commands can pipe the output of a command directly into the next one, avoiding the extra I/O overhead in Grunt.

Probably the biggest disadvantage to npm is the fact that Bash doesn't play well with Windows environments. Open source projects using `npm run` might run into issues when people try to fiddle with them on Windows. In a similar light, Windows developers will try to use alternatives to npm. That drawback pretty much rules out npm for projects that need to run on Windows.

Gulp, another build tool, presents similarities to both Grunt and npm, as you'll dis- cover in a moment.

Gulp: the streaming build tool

Gulp is similar to Grunt in that it relies on plugins and it's cross-platform, supporting Windows users as well. Gulp is a code-driven build tool, in contrast with Grunt's declarative approach to

task definition, making your task definitions a bit easier to read. Gulp is also similar to `npm run` in that it uses Node streams to read files and pipe data through functions that transform it into output that will end up written to disk. This means Gulp doesn't have the disk-intensive I/O issues you may observe when using Grunt. It's also faster than Grunt for the same reason: less time spent in I/O.

The main disadvantage to using Gulp is that it relies heavily on streams, pipes, and asynchronous code. Don't get me wrong; if you're into Node, that's definitely an advantage. But the issue with those concepts is that unless you and your team are well versed in Node, you'll probably run into issues dealing with streams if you have to build your own Gulp task plugins.

Gulp

There are a few things that are great about Gulp:

- High-quality plugins are readily available.
- Code-driven means your Gulpfile will be easier to follow than a configuration- driven Gruntfile.
- Faster than Grunt because it uses stream pipes rather than read and write to disk every time.
- Supports cross-platform development, the way Grunt does.

Gulp has drawbacks as well:

- It might be hard to learn if you don't have experience with Node.
- Developing quality plugins is hard for similar reasons.
- All of your team (current members and prospects) should be comfortable with streams and asynchronous code.
- The task dependency system leaves much to be desired.

When working in teams, Gulp isn't as prohibitive as npm. Most of your front-end team probably knows JavaScript, although chances are they're not that fluent in Bash script- ing, and some of them may be using Windows! That's why I usually suggest keeping `npm run` to your personal projects and maybe using Gulp in projects where the team is comfy with Node, and Grunt everywhere else. That's my personal opinion; figure out what works best for you and your team. Also, you shouldn't constrain yourself to Grunt, Gulp, or `npm run` because those tools work for me. Do research and maybe you'll find a tool that you like even better than those three.

Let's walk through several examples to get a feel for what Gulp tasks look like.

RUNNING TESTS IN GULP

Gulp is similar to Grunt in its conventions. In Grunt there's a `Gruntfile.js` file, used to define your build tasks, and in Gulp the file needs to be named `Gulpfile.js` instead. The

other minor difference is that in the case of Gulp, the CLI is contained in the same package as the task runner, so you have to install the `gulp` package from npm both locally and globally:

```
touch Gulpfile.js
npm install -g gulp
npm install --save-dev gulp
```

To get started, I'll create a Gulp task to lint a JavaScript file, using JSHint the way you've already seen with Grunt and `npm run`. In the case of Gulp, you have to install the `gulp-jshint` Gulp plugin for JSHint:

```
npm install --save-dev gulp-jshint
```

Now that you're fully equipped with the CLI that you globally installed, the local `gulp` installation, and the `gulp-jshint` plugin, you can put together the build task to run the linter. To define build tasks with Gulp, you have to write them programmatically in the `Gulpfile.js` file.

First, use `gulp.task`, passing it a task name and a function. The function contains

all of the code necessary to run that task. Here you should use `gulp.src` to create a read stream into your source files. You can provide the paths to individual files, or use a globbing pattern such as the ones you've seen in your experiences learning about Grunt. That same stream should be piped into the JSHint plugin, which you can con- figure or use with the defaults it comes with. Then all you have to do is pipe the results of the JSHint task through a reporter and have it print the results to your terminal. All of what I described results in the following Gulpfile:

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

gulp.task('test', function () {
  return gulp
    .src('./sample.js')
    .pipe(jshint())
    .pipe(jshint.reporter('default'));
})
;
```

By returning the stream, Gulp knows to wait until data stops flowing through it.

I should also mention that you're returning the stream so Gulp understands that it should wait for the data to stop flowing before it considers the task completed. You can use a custom JSHint reporter to make the output more concise and easier to read by humans. JSHint reporters don't need to be Gulp plugins, so you can use `jshint- stylish` for example. Let's install it locally:

```
npm install --save-dev jshint-stylish
```

The updated Gulpfile should look like the following code. It'll load the `jshint- stylish` module to format the reporting output.

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

gulp.task('test', function () {
  return gulp
    .src('./sample.js')
      .pipe(jshint())
      .pipe(jshint.reporter('jshint-stylish'));
})
;
```

You're done! That's all you have to do to declare a Gulp task named `test`. It can be run using the following command, provided you installed the `gulp` CLI globally:

```
gulp test
```

That was a trivial example. You can pipe the output of the JSHint linter through a reporter that will print the results of the linting test. You can also write output to disk using gulp.dest, which creates a write stream. Let's step through another build task.

BUILDING A LIBRARY IN GULP

To get started, let's do the bare minimum—read from disk with `gulp.src` and write back to disk piping the contents of the source file into `gulp.dest`, effectively copying the file into another directory:

```
var gulp = require('gulp');
gulp.task('build', function () {
  return gulp
    .src('./sample.js')
    .pipe(gulp.dest('./build'));
})
;
```

Copying the file is nice, but it doesn't minify its contents. To do that, you have to use a Gulp plugin. In this case you can use `gulp-uglify`, a plugin for the popular UglifyJS minifier:

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

gulp.task('build', function () {
  return gulp
    .src('./sample.js')
    .pipe(uglify())
    .pipe(gulp.dest('./build'));
})
;
```

As you probably realized, streams let you add more plugins while only reading and writing to disk once. As an example, let's pipe through `gulp-size` as well, which will calculate the size of the contents in the buffer and print that to the terminal. Note that if you add it before Uglify then you get the unminified size, and if you add it after, you get the minified size. You could also do both!

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

gulp.task('build', function () {
  return gulp
    .src('./sample.js')
    .pipe(uglify())
    .pipe(size())
    .pipe(gulp.dest('./build'));
})
;
```

To reinforce the point on the ability to add or remove pipes as needed, let's add one last plugin. This time you'll use `gulp-header` to add license information to the mini- fied piece of code, such as the name, the package version, and the license type. To run the example shown in the following listing, enter `gulp build` in your command line.

Listing 3 Using `gulp-header` to add license information

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var size = require('gulp-size');
var header = require('gulp-header');
var pkg = require('./package.json');
var info = '// <%= pkg.name %>@v<%= pkg.version %>, <%= pkg.license %>\n';

gulp.task('build', function () {
  return gulp
    .src('./sample.js')
    .pipe(uglify())
    .pipe(header(info, { pkg : pkg }))
    .pipe(size())
    .pipe(gulp.dest('./build'));
})
;
```

As in Grunt, in Gulp you can define flows by passing in an array of task names to `gulp.task`, instead of a function. The main difference between Grunt and Gulp in this regard is that Gulp executes these dependencies asynchronously, while Grunt exe- cutes them synchronously.

```
gulp.task('build', ['build-js', 'build-css']);
```

In Gulp, if you want to run tasks synchronously you have to declare a task as a depen- dency and then define your own task. All dependencies are executed before your task starts.

```
gulp.task('build', ['dep'], function () {
  // here goes the task that depends on 'dep'
})
;
```

If you take anything away from this overview, it should be that it doesn't matter which tool you use, as long as it allows you to compose the build flows you need in a way that doesn't make you work too hard for it.