How to Pick your Build Tool

By **Nico Bevacqua**, author of *JavaScript Application Design*

Committing to a build technology is hard. It's an important choice and you should treat it as such. In this article, based on the Appendix from *JavaScript Application Design*, you'll learn about three build tools used most often in front-end development workflows. The tools covered are Grunt, the configuration-driven build tool; npm, a package manager that can also double as a build tool; and Gulp, a code-driven build tool that's somewhere in between Grunt and npm.

Deciding on a technology is always hard. You don't want to make commitments you won't be able to back out of, but eventually you'll have to make a choice and go for something that does what you need it to do. Committing to a build technology is no different in this regard: it's an important choice and you should treat it as such.

There are three build tools I use most often in front-end development workflows. These are: Grunt, the configuration-driven build tool; npm, a package manager that can also double as a build tool; and Gulp, a code-driven build tool that's somewhere in between Grunt and npm. In this article, I'll lay out the situations in which a particular tool might be better than the others.

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 that 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.

There's a plugin base large enough that you'll rarely find yourself needing to develop your own build tasks, which also enables you and your team to rapidly develop a build process, which 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,” or “grunt-ec2,” to name a few.

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. Once 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 which belong to the same task individually, so that you're able to compose the flow in the right order. Since tasks are configured declaratively, you'll also have a hard time figuring out the order in which tasks get executed.

Besides that, your team should be dedicated to writing maintainable code when it comes to your builds as well, and in the case of Grunt that means maintaining separate files for the configuration of each task, or at least for each of the build flows that your team uses.

Grunt in a pinch

In short, Grunt has the following list of 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 with most teams

There are a few drawbacks to using Grunt, as well.

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

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

npm as a build tool

In order 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 manifest. The name of the property will be used as the task name, and the value will be the command you want to execute. The example shown below uses the JSHint command-line interface to run a linter through our JavaScript files and check for errors. You can run any shell command that you need.

```
{
  "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 following command.

```
npm run test
```

Note that npm provides shortcuts for specific task names. In the case of “test,” you could simply do “npm test” and omit the “run” verb. You can compose build flows by chaining “npm run” commands together in your script declarations.

```
{
  "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 could also schedule tasks as background jobs, making them asynchronous. Suppose we have the following package file, where we'll just copy a directory in our JavaScript build flow, and compile an Stylus stylesheet during our 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.

```
{
  "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 might need a Node package like stylus, or jshint as we saw in the last couple of examples. These dependencies should be installed through npm.

Installing NPM task dependencies

The JSHint CLI is not necessarily available in your system, and there are two ways you could go about installing it. If you're looking to use the tool directly from your command-line, and not in an “npm run” task, then you should install it globally, using the -g flag as shown below.

```
npm install -g jshint
```

However, if you're using the package in an npm run task, then you should be adding it as a devDependency, as shown below. That'll allow 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-devjshint
```

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

Using Shell scripts in NPM tasks

Below is an example 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 were to place that script in a file named “emoji” at the root of our 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.

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 thus 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 are already using to manage dependencies, and your package.json manifest, where dependencies and your build commands are listed. Since 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. This means that most Grunt tasks read from disk, and then write to disk. If you have several tasks working on the same files, then chances are the file is going to be read from disk multiple 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 all that well with Windows environments. This means that open source projects using “npm run” might run into issues when people try to fiddle with them on Windows. In a similar light, it also means that Windows developers will try and use alternatives to npm, instead. That drawback pretty much rules out npm for projects that need to be able to run on Windows.

Gulp, another build tool, presents similarities with both Grunt and npm, as you'll discover in a moment.

Gulp, the streaming build tool

Gulp is similar to Grunt in that it relies on plugins and its 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 being written to disk. This means that Gulp doesn't have the disk-intensive I/O issues that you may observe in using Grunt. It's also faster than Grunt for the same reason, less time spent in I/O.

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

Gulp at a glance

There's quite a few things that are great about Gulp.

- High quality plugins 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, just like Grunt

There's some drawbacks as well, of course.

- It might be hard to learn if you don't have some 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
- Task dependency system leaves much to be desired

When working in teams, Gulp is not as prohibitive as npm, because most of your front-end team probably knows JavaScript, while chances are they're not that fluent in Bash scripting, and some of them may be using Windows! That's why I usually suggest to keep npm run to your personal projects, maybe use Gulp in projects where the team is comfy with Node, and Grunt everywhere else. Of course, that's my personal appreciation, you should think for yourself and figure out what works best for you and your team. Also, you shouldn't constrain yourself to Grunt, Gulp, or npm run just because those are the tools that work for me. Try and do a little research, and maybe you'll find a tool that you like even better than those three.

Let's walk through some examples to get a feel of how Gulp tasks look.

Running tests in Gulp

Gulp is quite 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'll 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 just like you've already seen with Grunt and npm run. In the case of Gulp, you'll 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'll have to write them programmatically in the Gulpfile.js file.

First off, you'll 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, using a globbing pattern like the ones you've seen in our experiences learning about Grunt. That same stream should be piped into the JSHint plugin, which you can configure or just use with the defaults it comes with. Then all you'd 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've just described results in the Gulpfile presented below.

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

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

I should also mention that I'm returning the stream so that Gulp understands that it should wait for the data to stop flowing before it considers the task to be completed. You could use a custom JSHint reporter in order to have the output be a bit more concise, and thus easier to read by humans. JSHint reporters don't need to be Gulp plugins, so you could use “jshint-stylish” for example. Let's install it locally.

```
npm install --save-devjshint-stylish
```

The updated Gulpfile should look as shown below. It'll load the “jshint-stylish” module to format the reporting output.

```
var gulp = require('gulp');
varjshint = 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 there is to declaring a Gulp task named “test.” It can be run using the command below, provided that you installed the gulp CLI globally.

```
gulp test
```

That was quite a trivial example. Just as well as you can pipe the output of the JSHint linter through a reporter that will print the results of the linting test, you could also write output to disk by 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 just 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 quite minify its contents. To do that, you'll have to use a Gulp plugin. In this case you could use “gulp-uglify,” a plugin for the popular UglifyJSminifier.

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

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

As you've probably realized, streams enable you to 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 that are in the buffer, and print that to the terminal. Note that if you add it before Uglify then you'd get the unminified size, and if you add it after, you'll get the minified size. You could also do both!

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

```
});
```

Just to reinforce the point on composeability, being able to add or remove pipes as needed, let's add one last plugin. This time I'll use “gulp-header” to add some license information to the minified piece of code, such as the name, the package version, and the license type.

```
var gulp = require('gulp');
varuglify = require('gulp-uglify');
var size = require('gulp-size');
var header = require('gulp-header');
varpkg = 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'));
});
```

Just like 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 will execute these dependencies asynchronously, while Grunt executes them synchronously.

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

In Gulp, if you want to run tasks synchronously you'll have to declare a task as a dependency 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 article, have that 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.