Tools

Package Managers, Build Tools

December 2022

Agenda

1. INTRO
2. PACKAGE MANAGERS
3. TASK RUNNERS
4. MODULE BUNDLERS
5. TRANSPILERS, LINTERS

What are JS tools?

JavaScript continues to be the world's most popular programming language.

The popularity of the language comes in part from the many JavaScript tools that make programming easy and enjoyable. This is a list of well-known and popular tools for JavaScript, placed into categories that define important parts of the development process.

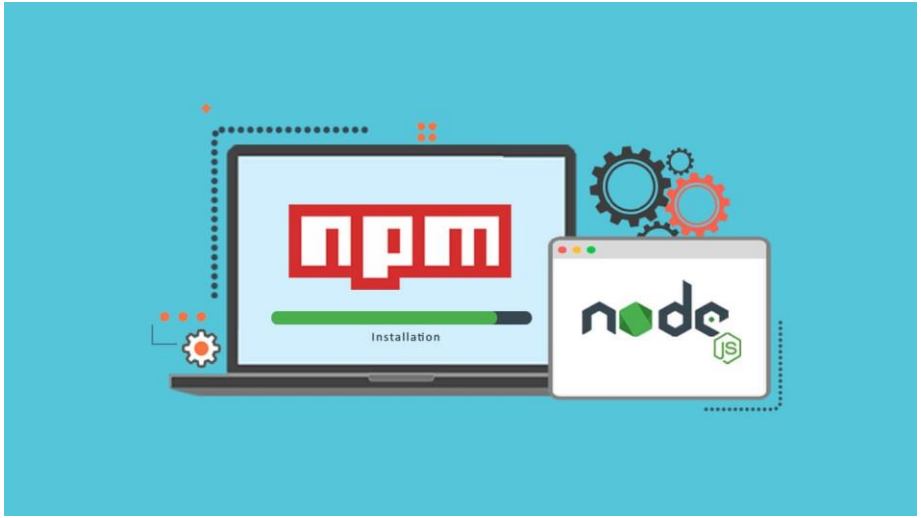
PACKAGE MANAGERS

What is package manager?

A package manager is a piece of software that lets you manage the dependencies (external code written by you or someone else) that your project needs to work correctly.

The package.json is a file that keeps track of all your dependencies (the packages to be managed). It also contains other metadata about your project.

Mentions: Lock File, Flat versus Nested Dependencies, Determinism vs Non-determinism



Dependency management

Package managers **simplify installing and updating project dependencies**, which are libraries such as: jQuery, Bootstrap, etc. – everything that is used on your site and isn't written by you.

Browsing all the library websites, downloading and unpacking the archives, copying files into the projects — all of this is replaced with a few commands in the terminal.

«Any application that can be written in JavaScript, will eventually be written in JavaScript»

Jeff Atwood - founder Stack Overflow

Yarn vs Npm

NPM

- Advantages
  - Is default and as good as Yarn in 2019
  - NVM & co
  - Lerna > Workspaces
  - Npm audit & security
  - Easy to use, especially for developers used to the workflow of older versions.
  - Local package installation is optimized to save hard drive space.
  - The simple UI helps reduce development time.
- Disadvantages
  - The online NPM registry can become unreliable in case of performance issues. This also means that NPM requires network access to install packages from the registry.
  - Despite a series of improvements across different versions, there are still security vulnerabilities when installing packages.
  - Command output can be difficult to read.

YARN

- Advantages
  - Ultra Fast
  - Mega Secure
  - Offline Mode
  - Deterministic
  - Network Performance
  - Flat Mode
  - Network Resilience
  - Workspaces
- Disadvantages
  - Yarn doesn't work with Node.js versions older than version 5.
  - Yarn has shown problems when trying to install native modules.

NPM

```
├── node_modules
├── package.json
└── package-lock.json (>5.0.0)
```

```
npm init
npm install
npm install -g <package_name>
npm install --save <package_name>
npm install --save-dev <package_name>
npm search <package_name>
npm update <package_name>
npm uninstall <package_name>
npm run <script_name>
npm -v
```

package.json

```
{
  "name": "Test",
  "version": "1.0.0",
  "main": "index.js",
  "repository": {},
  "author": "AG",
  "license": "MIT",
  "dependencies": {
    "babel-core": "^6.22.1",
    "lodash": "^4.17.4"
  },
  "devDependencies": {
    "karma": "^1.4.0"
  }
}
```

Yarn

- node_modules
- package.json
- yarn.lock

```
yarn init
yarn install
yarn add <package> [--dev]
yarn upgrade <package>
yarn remove <package>
yarn --version
```

package.json

```
{
  "name": "Test",
  "version": "1.0.0",
  "main": "index.js",
  "repository": {},
  "author": "AG",
  "license": "MIT",
  "dependencies": {
    "babel-core": "^6.22.1",
    "lodash": "^4.17.4"
  },
  "devDependencies": {
    "karma": "^1.4.0"
  }
}
```

PACKAGE.JSON DEPENDENCIES

Specifying version ranges:

- version - must match version exactly
- >version - must be greater than version
- >=version - must be greater than version or equal
- <version - must be less than version
- <=version - must be less than version or equal
- ~version - approximately equivalent to version (only patch updates)
- ^version - compatible with version (minor and patch updates)

package.json

```
{
    "dependencies" : {
    "foo" : "1.0.0 - 2.9999.9999",
    "bar" : "⩾1.0.2 <2.1.2",
    "baz" : ">1.0.2 ⩽2.3.4",
    "boo" : "2.0.1",
    "qux" : "<1.0.0 || ⩾2.3.1 <2.4.5 || ⩾2.5.2 <3.0.0",
    "asd" : "http://asdf.com/asdf.tar.gz",
    "til" : "~1.2",
    "elf" : "~1.2.3",
    "two" : "2.x",
    "thr" : "3.3.x",
    "lat" : "latest",
    "dyl" : "file:../dyl"
  }
}
```

TASK RUNNERS

WHY DID THEY COME?

We need to regularly perform some simple but important tasks:

Compile

Use Sass for CSS authoring because of all the useful abstraction it allows

Optimize

Optimize your images to reduce their file size without affecting quality.

Concatenate

Work in a small chunks of CSS and JS and concatenate them for the production website

Minify

Compress your CSS and minify JS to make their file sizes as small as possible.

Task runners/managers

Task managers control build steps for application and provide automation during development and build processes.

In other cases npm script may be used for this purpose.

In cases of advanced build steps (that has dependencies on other steps) Gulp or Grunt will be more suitable.

Task runners – pipeline process

- Source

  Preparing js, html, css
  (minifying, concatenating) Concatenate, Uglify, So
  urceMaps

- Test

  Karma, Protractor, Mocha, Coverage

- Watch

  LiveReload, Rebuild, Serve

- Assets

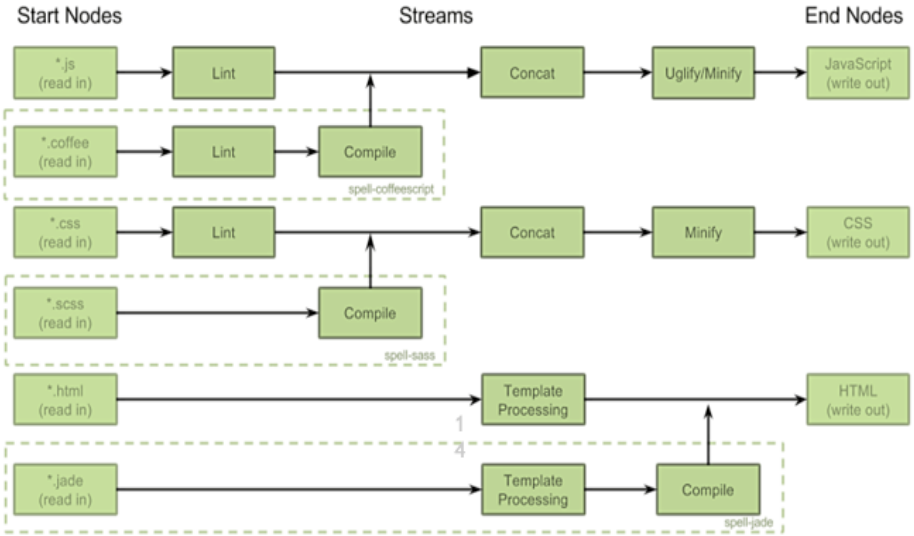
  Templates, CSS, HTML processing, Images optimiza
  tions

- Preprocess

  Compiling less/sass
  to css, LESS, SASS, Compass etc.

- Custom

  ChangeLog, Notifications, console.debug



INSTALL GULP

Install GULP globally:

```
$ npm install --global gulp
```

Install gulp in your project devDependencies:

```
$ npm install --save-dev gulp
```

GULP API

gulp.task

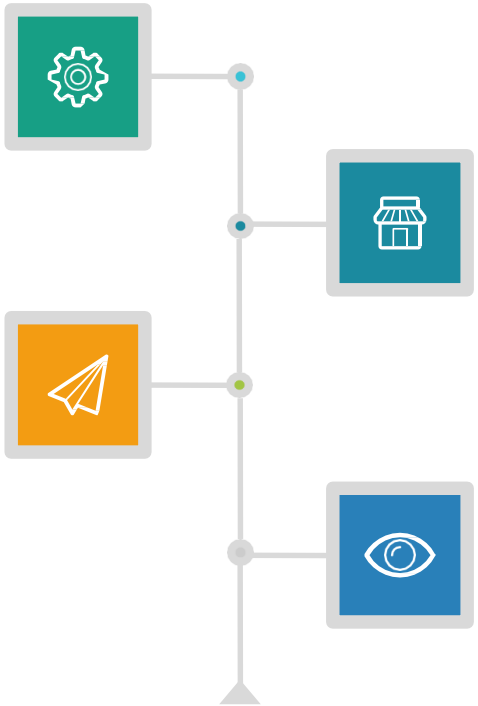
`gulp.task(name, [deps,], fn)`

Define a task with optional dependencies.

gulp.dest

`gulp.dest(folder)`

Save files from a stream to given directory.



gulp.src

`gulp.src(glob)`

Create a stream from given file system glob.

gulp.watch

`gulp.watch(glob, tasks)`

Run a task when one of the globbed files is changed.

SAMPLE GULPFILE

Create a gulpfile.js at the root of your project:

gulpfile.js

```
// Required modules
var gulp = require('gulp');
var uglify = require('gulp-uglify');

// Named tasks
gulp.task('scripts', function() {
// code here
});

// Watch tasks
gulp.task('watch', function() {  gulp.watch('app/js/**/*.js', ['scripts']);
});

// Default tasks
gulp.task('default',['scripts' , 'watch']);
```

Run gulp:

```
$ gulp [<task_name>]
```

LIVE RELOADING

Install:

```
$ npm install browser-sync --save-dev
```

Add it to gulpfile.js:

gulpfile.js

```
const gulp = require("gulp");
const browserSync = require("browser-sync").create();

gulp.task('webserver', function () {
  browserSync.init(browserSyncConfig);
});
```

JS Pipeline

Install plugins:

```
$ npm install gulp-eslint gulp-babel babel-core
gulp-concat gulp-uglify gulp-sourcemaps gulp-
browsersync --save-dev
```

Add plugins to gulpfile.js:

gulpfile.js

```
const gulp = require("gulp");
const uglify = require("gulp-uglify");
const sourcemaps = require("gulp-sourcemaps");
const concat = require("gulp-concat");
const eslint = require("gulp-eslint");
const babel = require("gulp-babel");
const browserSync = require("browser-sync").create();
const reload = browserSync.reload;
```

Write task:

gulpfile.js

```
gulp.task("js:build", function () {
  return gulp.src("./js/ *.js")
  .pipe(sourcemaps.init())
  .pipe(babel())
  .pipe(concat("all.js"))
  .pipe(uglify())
  .pipe(sourcemaps.write())
  .pipe(gulp.dest("./build/js"))
  .pipe(reload({stream: true}));
});

gulp.task("eslint", function () {
  return gulp.src("./js/ *.js")
  .pipe(eslint())
  .pipe(eslint.format())
  .pipe(eslint.failAfterError());
});
```

CSS PIPELINE

Install plugins:

```
$ npm install gulp-less gulp-cssmin gulp-
autoprefixer gulp-concat gulp-sourcemaps --save-dev
```

Add plugins to gulpfile.js:

gulpfile.js

```
const gulp = require("gulp");
const cssmin = require("gulp-minify-css");
const sourcemaps = require("gulp-sourcemaps");
const concat = require("gulp-concat");
const autoprefixer = require("gulp-autoprefixer");
const less = require("gulp-less");
const browserSync = require("browser-sync").create();
const reload = browserSync.reload;
```

Write task:

gulpfile.js

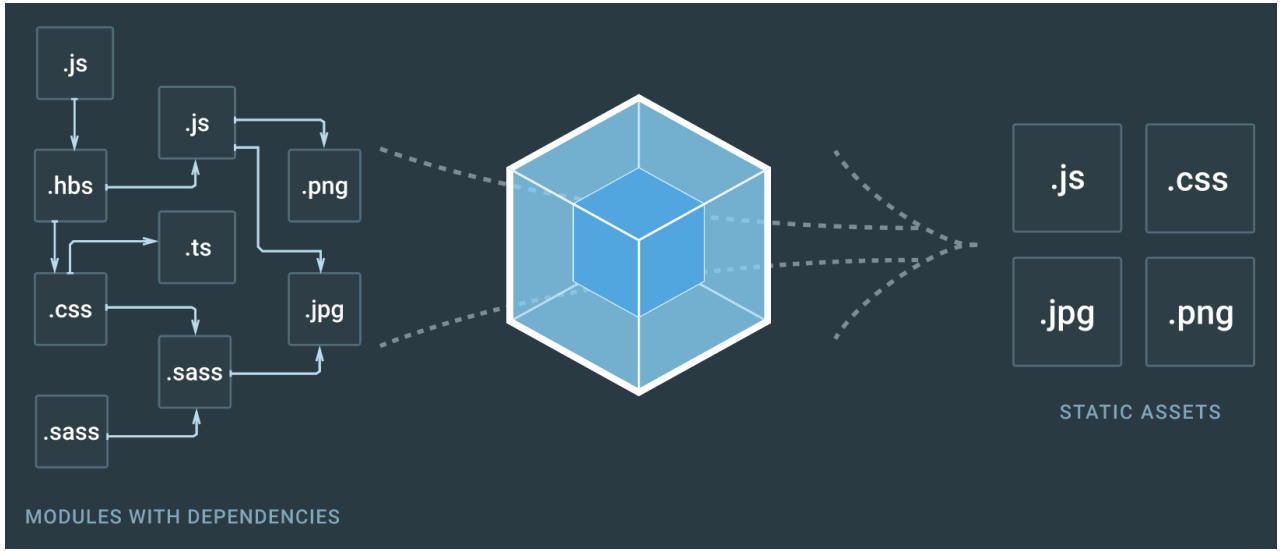
```
gulp.task("css:build", function () {
  return gulp.src("./less/ *.less")
  .pipe(sourcemaps.init())
  .pipe(less())
  .pipe(autoprefixer())
  .pipe(concat("all.css"))
  .pipe(cssmin())
  .pipe(sourcemaps.write())
  .pipe(gulp.dest("./build/css"))
  .pipe(reload({stream: true}));
});
```

MODULE BUNDLERS

WebPack

Webpack is used to compile JavaScript modules. Once installed, you can interface with webpack either from its CLI or API. If you're still new to webpack, please read through the core concepts and this comparison to learn why you might use it over the other tools that are out in the community.

It is a software that bundles all your JavaScript apps, as well as all kinds of different assets like images, font, and stylesheets. Supports ESM and CommonJS.



WebPack loaders

Loaders allow you to preprocess files as you require() or “load” them.

Loaders are kind of like “tasks” are in other build tools

code -> loaders -> plugins -> output

webpack.config.js

```
const NODE_ENV = process.env.NODE_ENV || 'development';
const webpack = require('webpack');

module.exports = {
  entry: './main.js',
  output: {
    filename: 'bundle.js',  library: 'bundle',  path: './dist'
  }, //for development
  watch: NODE_ENV === 'development',
  devtool: NODE_ENV === 'development' ? 'source-map' : null,
  plugins: [
    new webpack.DefinePlugin({
      NODE_ENV: JSON.stringify(NODE_ENV)
    }),
    new webpack.UglifyJSPlugin()
  ],
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
};
```

WebPack loaders

Basically loaders allow you to do a number of things like transform files from a different language to javascript, or inline images as data URLs. Loaders even allow you to do things like import CSS files directly from your JavaScript modules.

This is how we use loaders:

webpack.config.js

```
{
  test: /\.css$/,
  use: [“style-loader”, “css-loader”],
}
```

Here, we are testing if the file is a css file, and if it is, we are using the css-loader and the style-loader to transform the file before bundling it (webpack applies the loaders in order from right to left)

The way a loader gets matched against a resolved file can be configured in multiple ways, including by file type and by location within the file system.

Webpack final config example

- Entry
- Output
  - Path
  - Filename, with pattern approach
- Module
  - Rules, array of loaders
- Plugins
- Resolve
  - Alias

```
const webpack = require("webpack");
module.exports = {
  // Where to start bundling
  entry: {
    app: "./entry.js",
  },
// Where to output
  output: {
    // Output to the same directory
    path: __dirname,
// Capture name from the entry using a pattern
    filename: "[name].js",
  },
// How to resolve encountered imports
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ["style-loader", "css-loader"],
      },
      {
        test: /\.js$/,
        use: "babel-loader",
        exclude: /node_modules/,
      },
    ],
  },
// What extra processing to perform
  plugins: [
    new webpack.DefinePlugin({ ... }),
  ],
// Adjust module resolution algorithm
  resolve: {
    alias: { ... },
  },
};
```

TRANSPILERS
LINTERS

Transpilers

Transpilers, or source-to-source compilers, are tools that read source code written in one programming language, and produce the equivalent code in another language. Languages you write that transpile to JavaScript are often called compile-to-JS languages, and are said to target JavaScript.

Anything you can write in JavaScript, you can write in CoffeeScript or TypeScript.

script.js

```
"use strict";

// Good 'ol JS
function printSecret ( secret ) {
    console.log(`${secret}. But don't tell anyone.`);
}

printSecret("I don't like CoffeeScript");
```

script.ts

```
"use strict";

// TypeScript -- JavaScript, with types and stuff
function printSecret ( secret : string ) {
    console.log("${secret}. But don't tell anyone.");
}

printSecret("I don't like CoffeeScript.");
```

script.coffee

```
"use strict"

# CoffeeScript
printSecret (secret) ⇒
    console.log '#{secret}. But don\'t tell anyone.'

printSecret "I don't like JavaScript."
```

Transpilers

Babel is a tool for transpiling (compiling) ES6/ES7 code to ECMAScript 5 code, which can be used today in any modern browser.

```
$ npm install --save-dev babel-cli babel-preset-es2015
babel-plugin-transform-async-to-generator
```

.babelrc

```
{
    "presets": ["es2015"],
    "plugins": ["transform-async-to-generator"]
}
```

Linters

Linters and static code analysis helps to maintain code style during development, find potential bugs and performance issues.

Pros

- Comes configured and ready to go (if you agree with the rules it enforces)

Cons

- JSLint doesn't have a configuration file, which can be problematic if you need to change the settings
- Limited number of configuration options, many rules cannot be disabled
- You can't add custom rules
- Undocumented features
- Difficult to know which rule is causing which error

Pros

- Most settings can be configured
- Supports a configuration file, making it easier to use in larger projects
- Has support for many libraries out of
- the box, like jQuery, QUnit, NodeJS,
- Mocha, etc.
- Basic ES6 support

Cons

- Difficult to know which rule is causing an error
- Has two types of option: enforcing
- and relaxing (which can be used to make JSHint stricter, or to suppress its
- warnings). This can make
- configuration slightly confusing
- No custom rule support

Pros

- Supports custom reporters, which can make it easier to integrate with other tools
- Presets and ready-made configuration files can make it easy to set up if you follow one of the available coding styles
- Has a flag to include rule names in reports, so it's easy to figure out which rule is causing which error
- Can be extended with custom plugins

Cons

- Only detects coding style violations. JSCS doesn't detect potential bugs such as unused variables, or accidental globals, etc.

Linters

Pros

- Flexible: any rule can be toggled, and many rules have extra settings
- that can be tweaked
- Very extensible and has many
- plugins available
- Easy to understand output
- Includes many rules not available in other linters, making ESLint more useful for detecting problems
- Best ES6 support
- Supports custom reporters

Cons

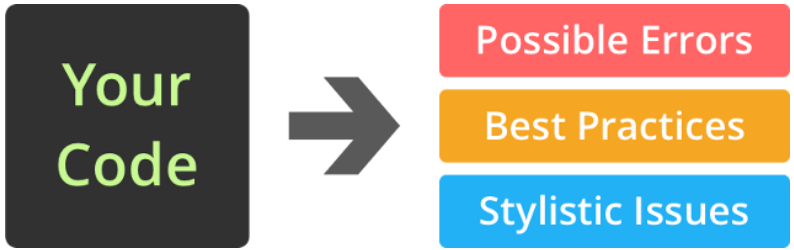
- Some configuration required
- Slow, but not a hindrance

Linters

Code linting is a way to increase code quality.

Linters like JSLint or JSHint can detect potential bugs, as well as code that is difficult to maintain.

Since linters are automated tools running them doesn't require manual work once they've been set up.



Linters

Given the fact that every developer has its own style in code writing, working with linter that warns you about rules your team has defined in your code style guide, could help your team keep the code maintainable and readable for all — present and future developers.

Example of a very common dispute in code style:

```
1
2   if (goodDeveloper === true) {
3       // This is the way you should write "if" statements
4   }
5
6   if (goodDeveloper === false)
7   {
8       // This is how evil developers are writing "if" statements
9   }
```

Documentation Software

Main documentation tools you should know:

- Swagger: Helps across the entire API lifecycle, from design to documentation. It's a set of rules and tools for describing APIs. It's language-agnostic and readable both by humans and machines.
- JSDoc: a markup language used to annotate JS source code files, which is then used to produce documentation in formats like HTML and RTF.

Debugging & Plugins & Extensions

Debugging tools make debugging less time-consuming and laborious, and they help the developer achieve more accurate results. A debugger tool can become your best friend in frustrating times.

- Chrome Developer Tools: A set of tools built directly into the Google Chrome browser, the Chrome Developer Tools have multiple utilities that help you debug JS code step by step.
- Node Inspect: Similar to the Chrome Developer Tools, but for when your app runs on Node.js.

Extensions

- Augury: is the most used Developer Tool extension for debugging and profiling Angular applications inside the Google Chrome and Mozilla Firefox browsers.

- Redux Devtools: Developer Tools to power-up Redux development workflow or any other architecture which handles the state change.

THANK YOU!