Tools

Package Managers, Build Tools

UA Resource Development Unit

2020

AGENDA

1. Intro
2. Package Managers
3. Task Runners
4. Module Bundlers
5. Transplilers, Linters

What are JS tools?

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

The popularity of JavaScript goes accompanied with a plethora of tools to make coding in JS easier. 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.

yarn

Bower

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

Jeff Atwood - founder Stack Overflow

NPM

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

Run in terminal

```
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
```

Run in terminal

```
yarn
init
yarn
insta
ll
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",
   "dependenci
   es": {
      "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, SourceMaps

Test
Karma, Protractor, Mocha, Coverage

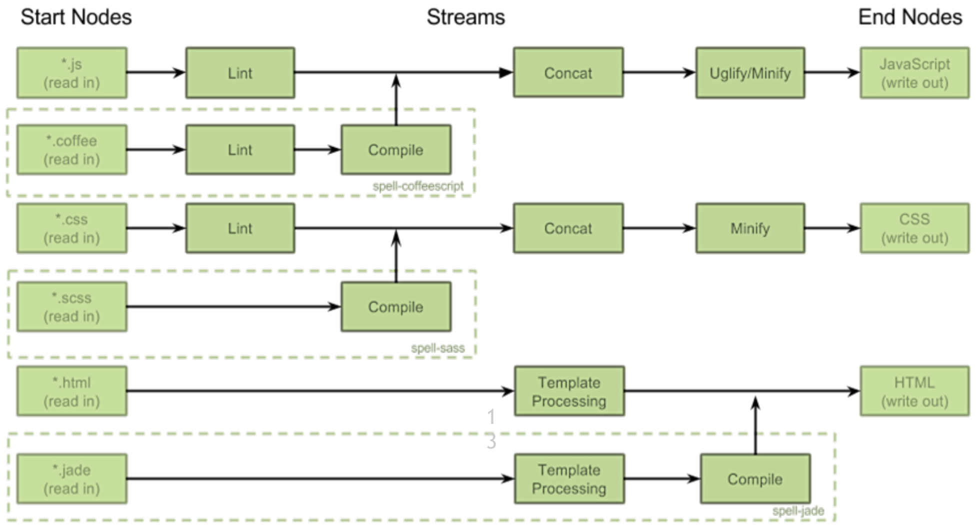
Watch
LiveReload, Rebuild, Serve

Assets
Templates, CSS, HTML processing, Images optimizations

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

Custom
ChangeLog, Notifications, console.debug

"Task to **automate** in your build pipeline"



Gulp

Gulp - streaming build system - system to describe arbitrary type tasks. Gulp plugins are installed and managed via npm, the Node.js package manager.

-By preferring code over configuration, gulp keeps things simple and makes complex tasks manageable.

- By enforcing strict plugin guidelines, we ensure that plugins stay simple and work as expected.

-Using node best practices and maintaining a minimal API surface, your build works exactly as you would imagine.

-Using the power of node streams, gulp gives you fast builds that don't write intermediary files to disk.

INSTALL GULP

Install gulp globally:

Run in terminal

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

Install gulp in your project devDependencies:

Run in terminal

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

GULP API

gulp.task

`gulp.task(name, [deps,], fn)`
Define a task with optional dependencies.

gulp.src

`gulp.src(glob)`
Create a stream from given file system glob.

gulp.dest

`gulp.dest(folder)`
Save files from a stream to given directory.

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

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
gulp.task('scripts', function() {
  // code here
});
gulp.task('watch', function() {
  gulp.watch('app/js/**/*.js', ['scripts']);
});
gulp.task('default',['scripts' , 'watch']);
```

Required Modules

Named tasks

Watch task

Default task

Run gulp

Run in terminal

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

LIVE RELOADING

Install:

Run in terminal

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

Add it to gulpfiles.js:

Gulpfile.js

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

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

JS PIPELINE

Install plugins:

Run in terminal

```
$ 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:

Run in terminal

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

Add it to gulpfiles.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;
```

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}));
});
```

TEMPLATES PIPELINE

Install:

Run in terminal

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

Add it to gulpfiles.js:

Gulpfile.js

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

Gulpfile.js

```
const JADE_LOCALS = {
      styles:
      ["css/all.css"]
      ,  scripts:
      ["js/all.js"]
};

gulp.task('html:build', function
      () {  return
      gulp.src(“./jade/*.jade")
            .pipe(jade({
            locals: JADE_LOCALS
            }))
            .pipe(gulp.dest(“./build"))
            .pipe(reload({stream: true}));
});
```

OPTIMIZING IMAGES

Install:

Run in terminal

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

Add it to gulpfiles.js:

Gulpfile.js

```
const gulp = require("gulp");
const imagemin = require("gulp-imagemin");

gulp.task("images", function ()
      {  return
       gulp.src('./images/*.jpg')
            .pipe(imagemin())
            .pipe(gulp.dest('./build/images/'
            ));
});
```

DEFAULT AND WATCH TASKS

Add it to gulpfiles.js:

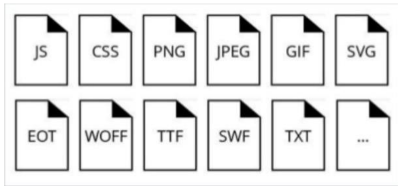
Gulpfile.js

```
const gulp = require("gulp");
const watch = require("gulp-watch");

gulp.task('watch', function(){
     gulp.watch(path.watch.jade,
     ['html:build']);
     gulp.watch(path.watch.less,
     ['css:build']);
     gulp.watch(path.watch.js,
     ['js:build', 'eslint']);
});

gulp.task('build', ['html:build', 'js:build', 'css:build', 'eslint`,
`images']);  gulp.task('default', ['build', 'webserver', 'watch']);
```

MODULE BUNDLERS

Modules bundlers

Assets



You can manage them manually, BUT....!

```
<!DOCTYPE html>
<html  lang="ru">
<head>
   <link rel="stylesheet" href=all.css">
   ...n
   <link rel="stylesheet“ href=some_css_n.css">
   <meta charset="utf-8">
</head>
<body>

<script  src="js/file1.js"></script>
<script  src="js/file2.js"></script>
...n
<script  src="js/file_n.js"></script>
<script  src="js/entry.js"></script>
</body>
</html>
```

browserify

rollup.js

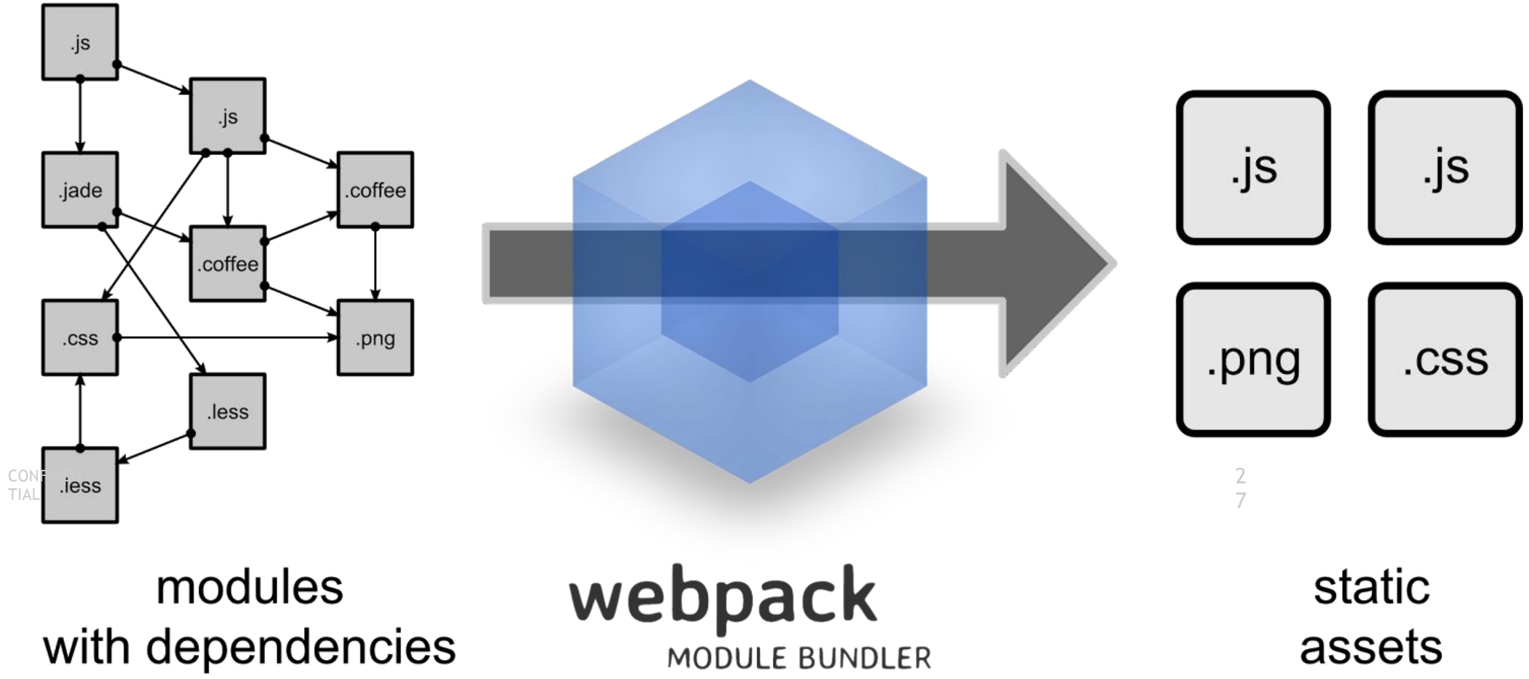
WebPack

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.

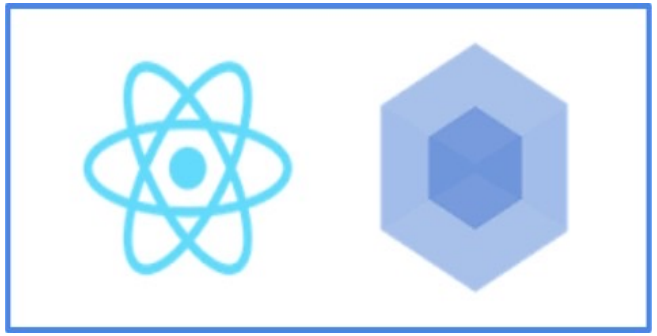
```
mkdir webpack-demo
cd webpack-demo
npm init -y
npm install webpack webpack-cli --save-dev
```

Webpack

Webpack: 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

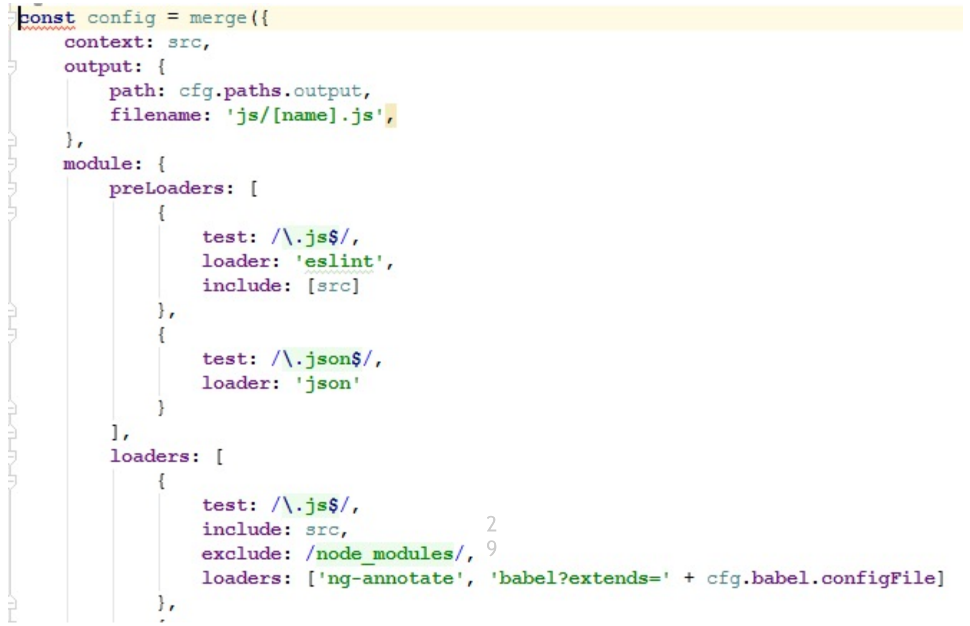


Webpack

Run in terminal

```
$  npm    install --save-dev webpack
```

webpack.config



```
const config = merge({
    context: src,
    output: {
        path: cfg.paths.output,
        filename: 'js/[name].js',
    },
    module: {
        preLoaders: [
            {
                test: /\.js$/,
                loader: 'eslint',
                include: [src]
            },
            {
                test: /\.json$/,
                loader: 'json'
            }
        ],
        loaders: [
            {
                test: /\.js$/,
                include: src,
                exclude: /node_modules/,
                loaders: ['ng-annotate', 'babel?extends=' + cfg.babel.configFile]
            },
```

Webpack

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:
   [
      newwebpack.DefinePlugin({
         NODE_ENV:JSON.stringify(NODE_ENV)
      }),
      newwebpack.UglifyJSPlugin()
   ],
   module: {
      loaders: [
         {
            test: /\.js$/,
            exclude: /(node_modules)/,
            loader: 'babel-loader'
         }
      ]
   }
};
```

WebPack. Loaders

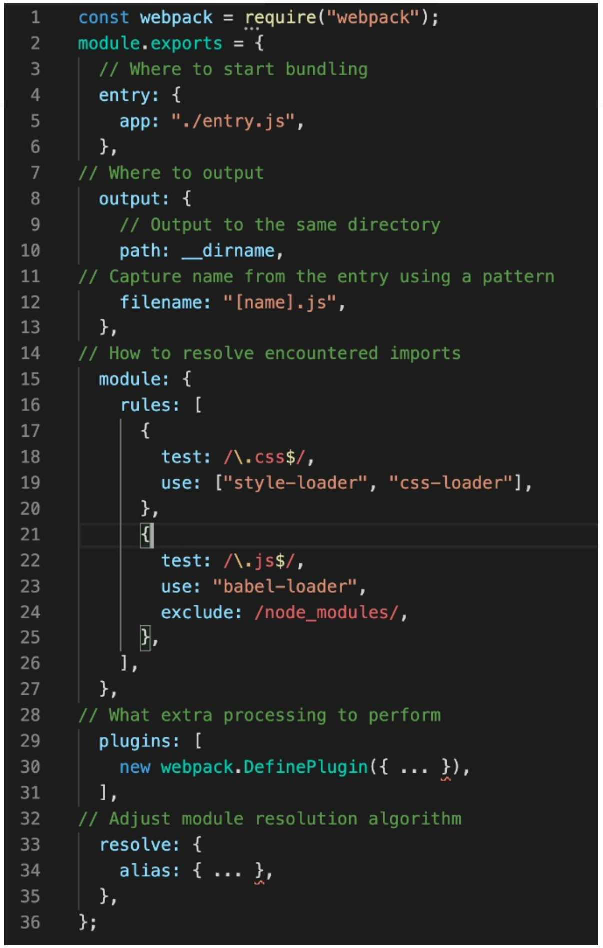
Basically loaders allow you to do a number of things like transform files from a different language to javascript, or inline images as data URLs. Loaders even allow you to do things like import CSS files directly from your JavaScript modules.
This is how we use loaders:

```
{
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

- Entry
- Output
  - Paht
  - 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.

script.js (plain javascript)

```
"use strict";
function printSecret(secret) {
   console.log(`${secret}. But don't tell
   anyone.`);
}
printSecret("I don't like  CoffeeScript");
```

script.ts (typescript)

```
"use strict";
function printSecret(secret: string){
   console.log("${secret}. But don't tell
   anyone.");
}
printSecret("I don't like  CoffeeScript.");
```

script.coffee (coffeescript)

```
"use strict"

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

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

Transpilers

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

Run in terminal

```
$ npm install    babel-cli    -g
$ npm install    babel-cli    --save-dev
                 //better
$ babel example.js      -o compiled.js
```

.babelrc

```
{
    "presets": ["es2015"],
    "plugins": []
}
```

All presets/plugins are separated packages. You need to install them as well.
E.g. *babel-preset-es2015*

Linters

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

JSLint

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

JS Hint

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

JSCS

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.

ESLint

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

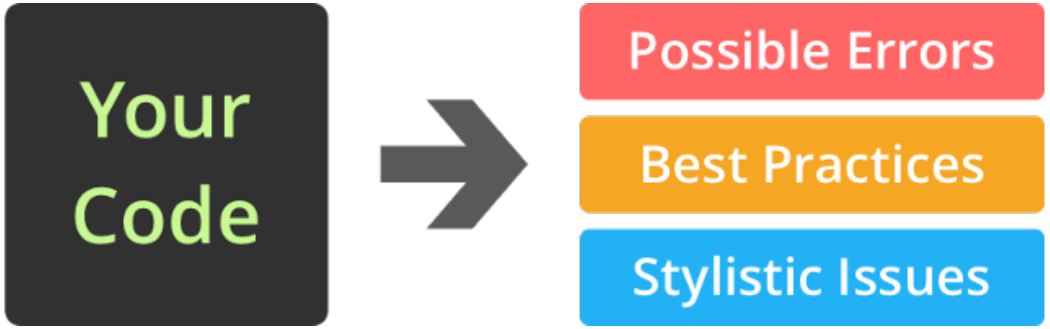
•Supports custom reporters

Cons

•Some configuration required

•Slow, but not a hindrance

Linters

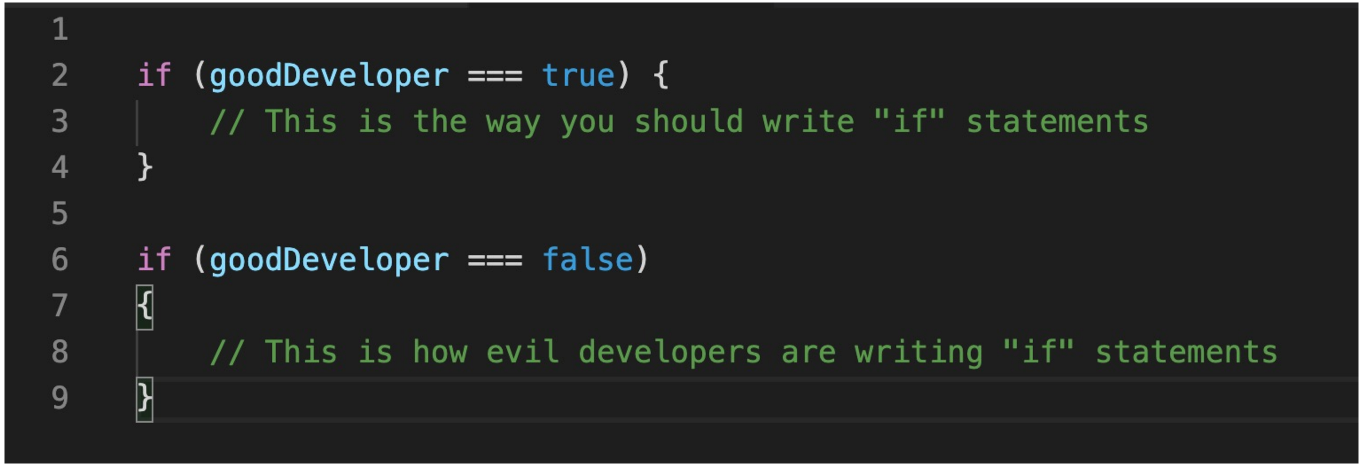


Code linting is a way to increase code quality.
Linters like JSLint or JSHint can detect potential bugs, as well as code that is difficult to maintain.
Since linters are automated tools running them doesn't require manual work once they've been set up.

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:



```

if (goodDeveloper === true) {
    // This is the way you should write "if" statements
}

if (goodDeveloper === false)
{
    // This is how evil developers are writing "if" statements
}
```

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.

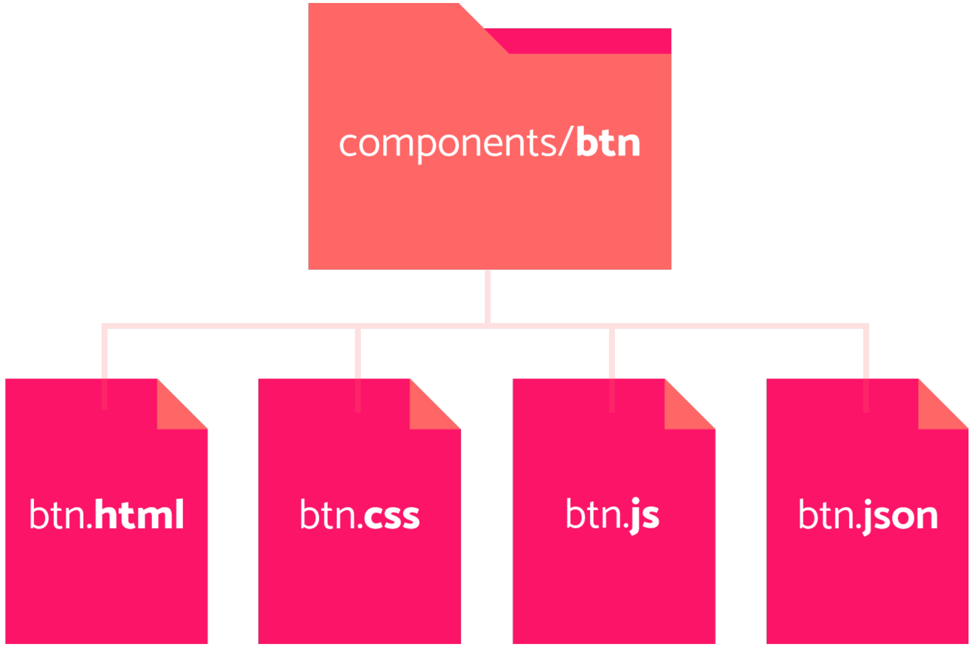
- Redux Devtools: Developer Tools to power-up Redux development workflow or any other architecture which handles the state change (see integrations).

Plugins

Google: top plugins for <IDE || Code Editor>

Project structure and tools

- Project structure
- Linters
- TypeScript
- Webpack
- Parcel



Project structure

Setting up a good front-end architecture is a fundamental step to start developing a web app or a website. Good practices and coding conventions are essential, but what about the structure? How can we conceive a good architecture that is maintainable in time? But most of all, where should we start from?

When I started thinking about the problem, I realized I needed a couple of things:

I wanted a multi-page project (a web app or website).

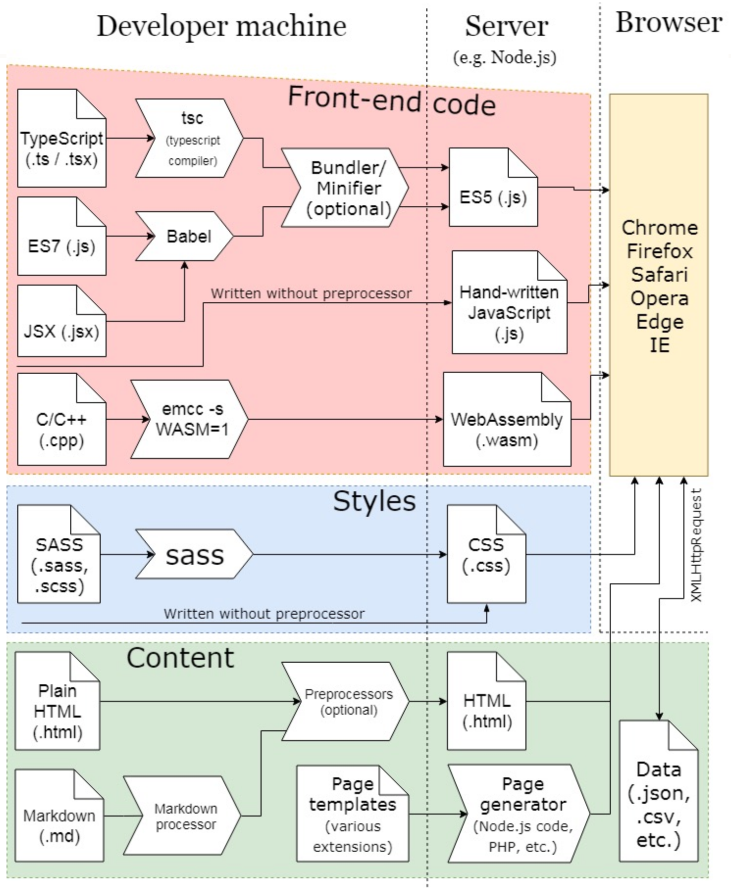
I wanted my project to support different screen sizes and resolutions, in other words: I wanted it to be responsive.

I wanted the final product to be maintainable.

I wanted the final product to be performant.

I wanted to reuse the same template for any future project.

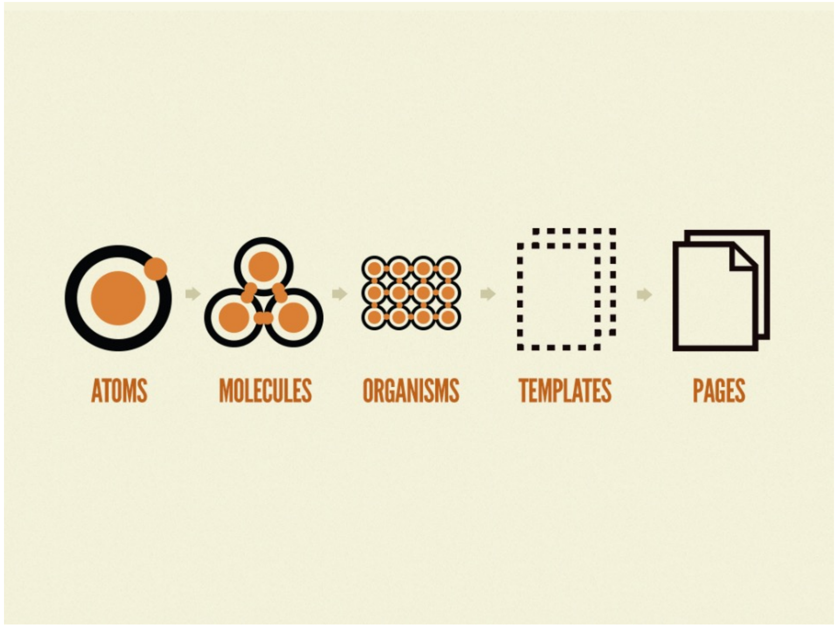
Project structure



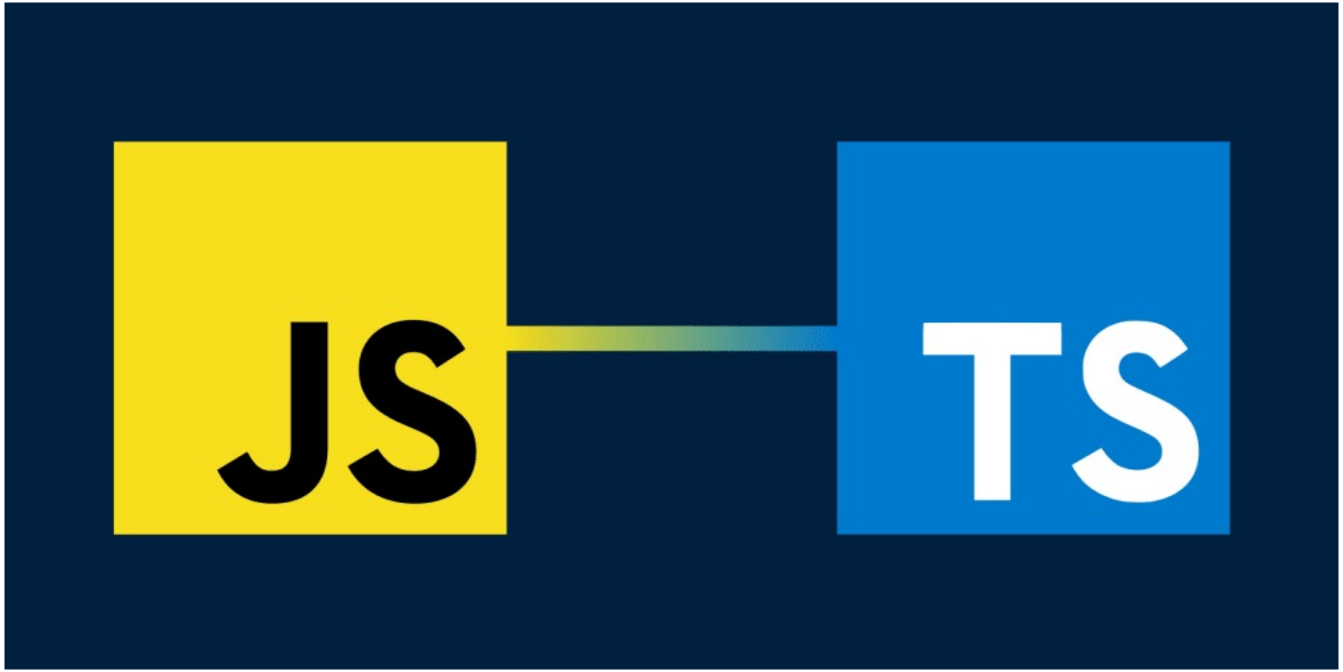
Project structure

Each component should only be concerned about itself, displaying the data passed to it, and dispatching appropriate actions when interacted with.

Split by components
Leading on from our first awesome site, we might decide to split out the HTML, CSS and Javascript into separate files to help manage our site's components. We can use tools such as SCSS, Twig and Browserify to handle compiling the files into a fully working site.



TypeScript



TypeScript. Setup

VS Code reportedly has TypeScript “language support” rather than a TypeScript **compiler**, so now we need to install the compiler.
There are two ways to install TypeScript with npm. Either use

- **npm install --global typescript**
- npm install --save-dev typescript

In your editor, type the following JavaScript code in greeter.ts:

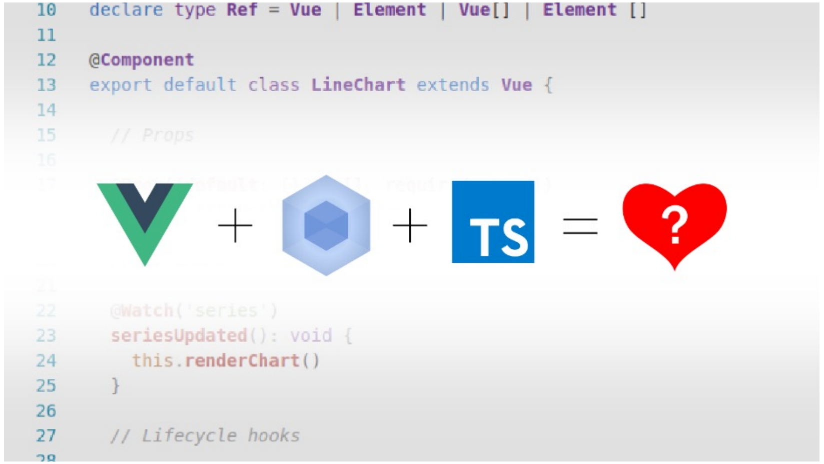
function greeter(person) { **return** "Hello, " + person; }
var user = "Jane User";
document.body.innerHTML = greeter(user);

We used a .ts extension, but this code is just JavaScript. You could have copy/pasted this straight out of an existing JavaScript app.
At the command line, run the TypeScript compiler:
tsc greeter.ts

The result will be a file greeter.js which contains the same JavaScript that you fed in

TypeScript.

TS could be used with different frameworks and tools



```
declare type Ref = Vue | Element | Vue[] | Element []

@Component
export default class LineChart extends Vue {

  // Props

  @Watch('series')
  seriesUpdated(): void {
    this.renderChart()
  }

  // Lifecycle hooks
```

Parcel

Building a basic app and want to get it up and running quickly? Use Parcel.
Building a library with minimal third-party imports? Use Rollup.
Building a complex app with lots of third-party integrations? Need good code splitting, use of static assets, and CommonJs dependencies? Use webpack.

PARCEL

Blazing fast, zero configuration web application bundler

Useful Links

- https://x-team.com/blog/essential-javascript-tools-2019/
- https://itnext.io/top-15-must-have-tools-for-javascript-developers-137fd825db65
- https://www.freecodecamp.org/news/javascript-package-managers-101-9afd926add0a/
- https://github.com/wbkd/webpack-starter
- https://github.com/cferdinandi/gulp-boilerplate
- https://www.toptal.com/front-end/webpack-browserify-gulp-which-is-better

Q&A

DRIVEN CANDID CREATIVE ORIGINAL INTELLIGENT EXPERT

UA Frontend Online LAB