LeanJS Documentation

Release 1.0.0

Romain Dorgueil

May 11, 2016

Contents

LeanJS is a clean, simple (yet featureful) opinionated NodeJS / React / Redux starter kit.

CHAPTER 1

Goals

- Setup a full-featured universal NodeJS / React / Redux project in a breaze.
- Don’t setup a hammer if you only need a screwdriver.
- Focus on value, get to code in seconds, not hours.

CHAPTER 2

Documentation

2.1 LeanJS

LeanJS is a clean, simple (yet featureful) opinionated NodeJS / React / Redux starter kit.

2.1.1 Goals

- Setup a full-featured universal NodeJS / React / Redux project in a breaze.
- Don't setup a hammer if you only need a screwdriver.
- Focus on value, get to code in seconds, not hours.

2.2 Getting started

2.2.1 Install

```
git clone https://github.com/hartym/LeanJS.git MyNextProject
cd MyNextProject
make install
```

2.2.2 Develop

- Launch the development server:

```
    make
```

- Open http://localhost:3001/ in your browser.
- Code.

2.2.3 Test

```
make lint
make test
```

2.2.4 Release

- Without docker, you can build a vanilla node project, just release the *build* directory once you've run:

```
make build
```

- With docker, it's even easier. Build a self-sufficient image with

```
make docker-build
```

2.3 Features

2.3.1 Included batteries

React

React is a modern templating engine that makes development of highly interractive user interfaces fun, fast, and "reactive".

In LeanJS, the library is used both browser-side and server-side, as our «isomorphic» rendering engine.

- https://facebook.github.io/react/

Redux

Redux is a «predictable state container» for JavaScript applications. In ohter terms, it's a library to manage the «state of your application», and more importantly, how do your application transition from one state to another. The library is very simple, but the underlying concepts are very powerful.

- http://redux.js.org/
- https://egghead.io/series/getting-started-with-redux

React Router

React Router is a complete routing solution for React. It works well in the «isomoprhic environment», abstracting the concept of «history» and «location» to be able to manage it the same way in the browser and on a server.

- https://github.com/reactjs/react-router
- https://github.com/reactjs/react-router-tutorial
- https://github.com/reactjs/react-router/tree/master/docs

Express (see *Server*)

Express, not to be presented anymore, is a minimalist web application development framework.

- http://expressjs.com/
- https://github.com/senchalabs/connect/blob/master/Readme.md

Bootstrap 4 and Font Awesome (see *Style*)

Bootstrap 4 and Font Awesome are so popular nowadays for prototypes (and even production apps) that it felt logic to package them in the boilerplate, pre-configured, ready to be extended and correctly setup to work both in dev and prod mode.

Removing them from the base setup is way easier than adding it, only a few lines to remove.

Attention, we include the plain vanilla SCSS versions of those libraries, with no wrappers (no, we don't ship with Bootact-Restrap). If you want one of those low value libs, feel free to have fun in your project.

SASS (see *Style*)

SASS is our CSS preprocessor of choice. It is already used by bootstrap, font-awesome and other well enginered fronted libraries, making it very easy to integrate and customize it.

2.3.2 Developer toolbox

On top of those features, that will be used the same way in development and in production mode, a few features are also there in development mode only.

GNU Make (see *Make*)

Build automation tool from 1977, readable by computer scientists, and language agnostic. It is the reference and only entry point of developer tools in LeanJS.

Docker (see *Docker*)

We use docker a lot to have reproductible builds of our apps. While it is not required that you use this feature, if you do, you can to build and run production image running *make docker-build docker-run*.

Webpack (see *Webpack*)

In development mode, webpack is used via the webpack-dev-middleware, which basically is a connect middleware that serve the always-hot version of your compiled assets.

In production mode, webpack is used to compile both frontend and backend javascript, into build. Just try to run *make build* and you'll get a clean and fresh production release in your build directory. You're only a *(cd build; npm install –production && node server)* away of running a lighting fast, pure, tool-free version of your project.

ESLint (see *Testing*)

make lint is your friend.

Help needed about the "right" coding standards for React/ES6/Node.

Karma, Tape and Istanbul (see *Testing*)

make test is your friend.

It will create a *coverage* directory with the matching reports.

Browsersync and Nodemon (see *Server*)

We use browser sync as a proxy for a few reasons.

- It allows to serve the vanilla express application on a different port and add the development tooling middlewares only on this proxy, making easier to debug tedious problems with server output.
- It has nice features for both cross-platform development and simulating catastrophic network conditions, because you know, your customer won't use your own laptop on this dual gigabit fiber channel directly plugged on the EU to NA backbone.

Hot Module Reload (see `reference/hot-module-reload`)

This is trendy, this is hot, and yet it can fuck up your brain because it does not work as expected.

In short, HMR is a way to swap module instances hen you change the files on disk. Save Foo.js, a watcher will see it, rebuild it with webpack, communicate that something changed to your browsers and try to hot swap the minimum subset of the module dependency tree that is needed to have the new module versions show up in the browsers. Magic happens, boom.

That's the theory. Practical notes, though.

Caveats: won't work with pure, stateless, function-based react components. won't work with router.

When in doubt, refresh anyway.

2.4 Recipes

Some commonly used code recipes.

- `recipes/bootstrap`
- `recipes/font-awesome`

2.5 Reference

Directory structure:

- bin: management and instrumentation scripts.
- build: production releases (out of version control).
- coverage: generated coverage data (out of version control).
- config: declarative configuration, including webpack configuration.
- src: application(s) source code.
- test: tests, using tape as a default.

(TODO: this is a work in progress, reference is lacking a lot of sections as of now).

2.5.1 Docker

Small helpers exist to help you build docker images from your project.

The docker image name that will get built is defined by the *$DOCKER_IMAGE* environment variable, and you can change its default value in *Makefile*.

To build an image, run:

```
make docker-build
```

You can customize target image name on the fly:

```
DOCKER_IMAGE=acme/foobar make docker-build
```

To run the image, exposing the port on docker server machine:

```
make docker-run
```

2.5.2 Make

We use the old school GNU Make to manage shell entry points in LeanJS. You may prefer to use Gulp, Grunt, or some other task manager, but we don't see any feature that we need and that Make does not provide.

It has the advantage to be understandable by non-javascript persons, including ops, and we use it to provide a language agnostic interface to our projects (we provide the same commands whether the project is written using javascript, python, ruby, java, php, ...).

The pros may be not very important for you, feel free to change the way you manage this.

Targets

start (default)

Usage: *make start* or *make*

Throws up a development server (see Server).

```
# Start a development server.
# You need to run "npm install" before that.
start:
        npm run start
```

build

Usage: *make build*

Compiles a production ready build in the *build* folder.

```
# Builds a production-ready release, under /build.
# It will be self contained, and it's the base of the docker image.
build:
        npm run build
```

install

Usage: *make install*

Wraps *npm install*, installs all project dependencies under *node_modules*, including development dependencies.

```
# Install dependencies (dev + prod)
install:
        npm install
```

2.5.3 Server

Development

To run the development server, make sure the project dependencies, including development dependencies, are installed (if in doubt, run *make install*), then fire up the development server by simply running *make*.

After a little while, you'll have three different servers running:

- On port 3000, the production-like express server will run. Thanks to nodemon, the server will be restarted whenever you change a file. The webpack managed assets won't be available though, so you should not try to develop looking at this port, it should only be useful to debug server-side rendering issues.
- On port 3001, Browsersync will serve the real application, by proxying the express server above. It will add some middlewares so your assets are available, and hot module replacement (HMR) will try to hot-swap frontend components that you change. It is not always possible, but it will try its best.
- On port 3002, Browsersync will serve its management interface. Go ahead and explore the features, as it's pretty cool.

Production

On production, you'll run the express server in a context where webpack precompiled all assets, making it effective.

To run it, you can *(make build; cd build; node server)*. Otherwise, you can simply *make build* and use the release you just build in the *build* directory to run it however you want. That could be docker (see Docker), but you can pretty much do it any way you want.

2.5.4 Style

You want your application to be beautiful, as a default, right? You don't want to spend huge amount of time setting it up, right?

We chosed a few defaults for that:

- SASS
- Bootstrap 4
- Font Awesome

Those are defaults, not requirements, but it allows us to provide a base framework to develop your prototypes as quickly as possible. Feel free to replace it with whatever you want as your project grows.

2.5.5 Testing

Tests are important. Even though you may want to deliberately skip some in your prototypes, knowing you can throw out a solid testing architecture in a snap is always great.

```
make lint
make test
```

After a *make test* run, you'll find coverage reports in the *coverage* folder.

2.5.6 Webpack

Webpack is a powerfull tool. We split the configuration (found in *config/webpack*) in three different files: *default.js*, *client.js* and *server.js*.

Both client and server configurations extend default, and if more targets (cordova, electron, browser extensions, ...) were to be added to the project, a similar file structure could be kept.

The loaders configuration has been split, for better readability and code reuse.

Default configuration

Default configuration define the common chunks of configuration that will be used whatever the target platform is.

```
import path from 'path'
import config from '../index'
import loaders from './loaders'

const AUTOPREFIXER_BROWSERS = [
  'Android 2.3',
  'Android >= 4',
  'Chrome >= 35',
  'Firefox >= 31',
  'Explorer >= 9',
  'iOS >= 7',
  'Opera >= 12',
  'Safari >= 7.1'
]

const defaultWebpackConfig = {
  // Input
  context: path.resolve(__dirname, '../../src'),

  // Output
  output: {
    path: path.resolve(__dirname, '../../build'),
    publicPath: '/',
    sourcePrefix: '  '
  },

  // Loaders
  module: { loaders },

  // is it the right place ? at least needed server side for HMR. Is it required if DEBUG == false?
  recordsPath: path.resolve(__dirname, '../../build/_records'),

  resolve: {
```

```
    root: path.resolve(__dirname, '../../src'),
    extensions: ['', '.webpack.js', '.web.js', '.js', '.jsx', '.json']
  },

  // Cache generated modules and chunks to improve performance for multiple incremental builds.
  cache: config.DEBUG,

  // Switch loaders to debug mode.
  debug: config.DEBUG,

  // TODO Use debug here to choose value?
  devtool: 'source-map',

  stats: {
    colors: true,
    reasons: config.DEBUG,
    hash: config.VERBOSE,
    version: config.VERBOSE,
    timings: true,
    chunks: config.VERBOSE,
    chunkModules: config.VERBOSE,
    cached: config.VERBOSE,
    cachedAssets: config.VERBOSE
  },

  plugins: [],

  sassLoader: {
    includePaths: [path.resolve(__dirname, '../node_modules')]
  },

  /* eslint-disable global-require */
  postcss (bundler) {
    return [
      require('postcss-import')({ addDependencyTo: bundler }),
      require('precss')(),
      require('autoprefixer')({ browsers: AUTOPREFIXER_BROWSERS })
    ]
  }
  /* eslint-enable global-require */
}

export default defaultWebpackConfig
```

Client configuration

Client configuration (may be renamed to browser at some point, we'll see) is in charge of transpiling the browser javascript so it can run in modern browsers.

```
import path from 'path'
import webpack from 'webpack'
import AssetsPlugin from 'assets-webpack-plugin'
import CompressionPlugin from 'compression-webpack-plugin'
import ExtractTextPlugin from 'extract-text-webpack-plugin'
import config from '../index'
import defaultConfig from './default'
```

```
const clientConfig = {
  ...defaultConfig,

  entry: './client.js',

  output: {
    ...defaultConfig.output,
    path: path.join(defaultConfig.output.path, 'public/'),
    filename: config.DEBUG ? '[name].js?[chunkhash]' : '[name].[chunkhash].js',
    chunkFilename: config.DEBUG ? '[name].[id].js?[chunkhash]' : '[name].[id].[chunkhash].js'
  },

  plugins: [
    ...defaultConfig.plugins,

    new webpack.DefinePlugin({ ...config, 'process.env.BROWSER': true }),

    // Emit a file with assets paths
    // https://github.com/sporto/assets-webpack-plugin#options
    new AssetsPlugin({
      path: path.resolve(__dirname, '../../build'),
      filename: 'assets.js',
      processOutput: (x) => `module.exports = ${JSON.stringify(x)};`
    }),

    // Add production plugins if we're doing an optimized build
    ...(!config.DEBUG ? [
      new ExtractTextPlugin('[name].[chunkhash].css', { allChunks: true }),
      new webpack.optimize.DedupePlugin(),
      new webpack.optimize.UglifyJsPlugin({
        compress: {
          // jscs:disable requireCamelCaseOrUpperCaseIdentifiers
          screw_ie8: true,
          // jscs:enable requireCamelCaseOrUpperCaseIdentifiers
          warnings: config.VERBOSE
        }
      }),
      new webpack.optimize.AggressiveMergingPlugin(),
      new CompressionPlugin()
    ] : [])
  ]
}

if (!config.DEBUG) {
  // Production: Replace loaders with "ExtractTextPlugin"
  const originalLoaders = clientConfig.module.loaders[1].loaders
  delete clientConfig.module.loaders[1].loaders
  clientConfig.module.loaders[1].loader = ExtractTextPlugin.extract(...originalLoaders)
}

export default clientConfig
```

Server configuration

Server configuration is in charge of transpiling code required to run the production-ready express server, to be directly run by the Node interpreter.

```
import webpack from 'webpack'
import config from '../index'

import defaultConfig from './default'

/**
 * This is Webpack configuration for server side javascript, aimed for a build
 * without intrumentation (like dev middleware or hot module reload).
 *
 * Instrumentation will be added by the runServer script, if needed.
 */
const serverConfig = {
  // Extends common configuration
  ...defaultConfig,

  // Entry point
  entry: './server.js',

  // Output a single server.js, under the build directory
  output: {
    ...defaultConfig.output,
    filename: 'server.js',
    libraryTarget: 'commonjs2'
  },

  // This will be run by node. Also used by our scripts to detect it is
  // the server-side configuration.
  target: 'node',

  // How do we take apart bundlable scripts and external dependencies that we
  // can load from filesystem?
  externals: [
    /^\.\/assets$/,
    function filter (context, request, cb) {
      const isExternal = request.match(/^[@a-z][a-z\/\.\-0-9]*$/i)
      cb(null, Boolean(isExternal))
    }
  ],

  node: {
    console: false,
    global: false,
    process: false,
    Buffer: false,
    __filename: false,
    __dirname: false
  },

  plugins: [
    ...defaultConfig.plugins,
    new webpack.DefinePlugin({ ...config, 'process.env.BROWSER': false }),
    new webpack.BannerPlugin('require("source-map-support").install();',
      { raw: true, entryOnly: false })
  ]
}

export default serverConfig
```

Loaders configuration

All loaders

```
import config from '../../index'
import javascriptLoader from './javascript'
import styleLoader from './style'

export default [
  javascriptLoader,
  styleLoader,
  { test: /\.json$/, loader: 'json-loader' },
  { test: /\.txt$/, loader: 'raw-loader' },
  {
    test: /\.(png|jpg|jpeg|gif|svg|woff|woff2)(\?v=[0-9]\.[0-9]\.[0-9])?$/,
    loader: 'url-loader',
    query: {
      name: config.DEBUG ? '[path][name].[ext]?[hash]' : '[hash].[ext]',
      limit: 10000
    }
  },
  {
    test: /\.(ttf|eot|wav|mp3)(\?v=[0-9]\.[0-9]\.[0-9])?$/,
    loader: 'file-loader',
    query: {
      name: config.DEBUG ? '[path][name].[ext]?[hash]' : '[hash].[ext]'
    }
  }
]
```

Javascript

```
import config from '../../index'
import path from 'path'

export default {
  test: /\.jsx?$/,
  loader: 'babel-loader',
  include: [
    path.resolve(__dirname, '../../../src'),
    path.resolve(__dirname, '../../../config'),
    path.resolve(__dirname, '../../../test'),
    path.resolve(__dirname, '../../../build/assets')
  ],
  query: {
    // https://github.com/babel/babel-loader#options
    cacheDirectory: config.DEBUG,

    // https://babeljs.io/docs/usage/options/
    babelrc: false,
    presets: [
      'react',
      'es2015',
      'stage-0'
    ],
    plugins: [
```

```
      'transform-runtime',
      ...config.DEBUG ? [] : [
        'transform-react-remove-prop-types',
        'transform-react-constant-elements',
        'transform-react-inline-elements'
      ]
    ]
  }
}
```

Style

```
import config from '../../index'

export default {
  test: /\.scss$/,
  loaders: [
    'style',
    `css?${JSON.stringify({
      // `sourceMap` is set to false because otherwise, there will be a problem with custom fonts
      // when using the development proxy.
      // See http://stackoverflow.com/questions/34133808/webpack-ots-parsing-error-loading-fonts
      sourceMap: false,
      // CSS Modules https://github.com/css-modules/css-modules
      // modules: true,
      localIdentName: config.DEBUG ? '[name]_[local]_[hash:base64:3]' : '[hash:base64:4]',
      // CSS Nano http://cssnano.co/options/
      minimize: !config.DEBUG
    })}!sass?${JSON.stringify({
      sourceMap: config.DEBUG
    })}`
  ]
}
```

2.6 (Not So) Frequently Asked Questions

2.6.1 «Why is there no subdirectories, and no strong choices in directory structure like I can find in *insert something here*?»

We think it's up to you to make such kind of choices. For example, if you have a good reason to separate what you'd call «components» and «containers» (a.k.a dumb components and presentation-less containers), you will for sure know it, and thus you don't need the boilerplate to do it for you.

We tryed to keed the base directory structure as simple as possible, and as unopinionated as possible.

2.6.2 I want to use CSS Modules, Relay, Passport, Firebase, Auth0, *insert your own here*, why isn't that included in the boilerplate?

We don't want to include it in the base project, as it could be more harm than good.

If you want/need it, the best option would be to write an integration recipe in this document, so it takes only a few minutes to add it to a blank project.

CHAPTER 3

Indices and tables

- genindex
- search