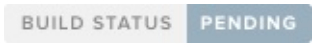
Table of Contents

> The boilerplate for making electron applications built with vue (pretty much what it sounds like).



Overview

The aim of this project is to remove the need of manually setting up electron apps using vue. electron-vue takes advantage of `vue-cli` for scaffolding, `webpack` with `vue-loader`, `electron-packager` or `electron-builder`, some of the most used plugins like `vue-router`, `vuex`, and so much more.

Check out the documentation here.

Things you'll find in this boilerplate...

- Basic project structure with a **single** `package.json` setup
- Detailed documentation
- Project scaffolding using vue-cli
- Ready to use Vue plugins (axios, vue-electron, vue-router, vuex)*
- Installed vue-devtools and devtron tools for development
- Ability to easily package your application using electron-packager or electron-builder*
- `appveyor.yml` and `.travis.yml` configurations for automated deployments with electron-builder*
- Ability to produce web output for browsers
- Handy NPM scripts
- Use of webpack and vue-loader with Hot Module Replacement
- Process restarting when working in electron's `main` process
- HTML/CSS/JS pre-processor support with vue-loader
- ES6 with `stage-0` by default
- Use of `babili` to remove the need of transpiling completely down to ES5
- ESLint (with support for `standard` and `airbnb-base`)*
- Unit Testing (with Karma + Mocha)*
- End-to-end Testing (with Spectron + Mocha)*

*Customizable during `vue-cli` scaffolding

Getting Started

This boilerplate was built as a template for `vue-cli` and includes options to customize your final scaffolded app. The use of `node@^7` or higher is required. electron-vue also officially recommends the `yarn` package manager as it handles dependencies much better and can help reduce final build size with `yarn clean`.

```
# Install vue-cli and scaffold boilerplate
npm install -g vue-cli
vue init simulatedgreg/electron-vue my-project

# Install dependencies and run your app
cd my-project
yarn # or npm install
yarn run dev # or npm run dev
```

Are you a Windows User?

Make sure to check out **A Note for Windows Users** to make sure you have all the necessary build tools needed for electron and other dependencies.

Wanting to use Vue 1?

Just point to the `1.0` branch. Please note that electron-vue has officially deprecated the usage of `vue@^1`, so project structure, features, and documentation will reflect those changes (**legacy documentation**).

```
vue init simulatedgreg/electron-vue#1.0 my-project
```

Next Steps

Make sure to take a look at the documentation. Here you will find useful information about configuration, project structure, and building your app. There's also a handy FAQs section.

Made with electron-vue

Take a look at some of the amazing projects built with electron-vue. Want to have your own project listed? Feel free to submit a pull request.

- **Surfbird**: A Twitter client built on Electron and Vue
- **Lulumi-browser**: Lulumi-browser is a light weight browser coded with Vue.js 2 and Electron
- **Space-Snake**: A Desktop game built with Electron and Vue.js.
- **Forrest**: An npm scripts desktop client
- **miikun**: A Simple Markdown Editor
- **Dakika**: A minute taking application that makes writing minutes a breeze
- **Dynamoc**: Dynamoc is a GUI client for dynamodb-local, dynalite and AWS dynamodb
- **Dockeron**: A dockeron project, built on Electron + Vue.js for Docker
- **Easysubs**: Download subtitles in a very fast and simple way
- **adminScheduler**: An application leveraging electron for cross platform compatibility, Vue.js for lightning fast UI and full-calendar.io to deliver a premium calendar interface.
- **Data-curator**: Share usable open data.

Getting Started

Scaffolding

This boilerplate was built as a template for vue-cli and includes options to customize your final scaffolded application. The use of `node@^7` or higher is required. electron-vue also officially recommends the `yarn` package manager as it handles dependencies much better and can help reduce final build size with `yarn clean`.

```
# Install vue-cli and scaffold boilerplate
npm install -g vue-cli
vue init simulatedgreg/electron-vue my-project

# Install dependencies and run your app
cd my-project
yarn # or npm install
yarn run dev # or npm run dev
```

On the subject of electron

Although optional, it is recommended to lock in your electron version after scaffolding your project. This helps prevent other developers working on the same project from developing on a different version. Electron makes releases quite often so features are always subject to change. More Info.

A note for Windows Users

If you run into errors during `npm install` about `node-gyp`, then you most likely do not have the proper build tools installed on your system. Build tools include items like Python and Visual Studio. Thanks to @felixrieseberg, there are a few packages to help simplify this process.

The first item we need to check is our npm version and ensure that it is not outdated. This can is accomplished using `npm-windows-upgrade`. If you are using `yarn`, then you can skip this check.

Once that is complete, we can then continue to setup the needed build tools. Using `windows-build-tools`, most of the dirty work is done for us. Installing this globally will in turn setup Visual C++ packages, Python, and more.

At this point things should successfully install, but if not then you will need a clean installation of Visual Studio. Please note that these are not direct problems with electron-vue itself (Windows can be difficult sometimes ¯_(ツ)_/¯).

Project Structure

When it comes to making electron apps, project structure is a little different. If you have used the official `vuejs-templates/webpack` setup before, then the structure should feel quite familiar. The documentation in this section attempts to explain a general overview of how the boilerplate works and the differences when your application is built.

Single `package.json` Setup

There was a time not too long ago where a two `package.json` setup was neccessary, but thanks to efforts from @electron-userland, both `electron-packager` and `electron-builder` now fully support a single `package.json` setup.

`dependencies`

These dependencies **will** be included in your final production app. So if your application needs a certain module to function, then install it here!

`devDependencies`

These dependencies **will not** be included in your final production app. Here you can install modules needed specifically for development like build scripts, `webpack` loaders, etc.

Installing Native NPM Modules

We need to make sure our native npm modules are built against electron. To do that, we can use `electron-rebuild`, but to make things simpler, it is highly recommended to use `electron-builder` for your build tool as a lot of these tasks are handled for you.

On the subject of the `main` process

During development you may notice `src/main/index.dev.js`. This file is used specifically for development and is used to install dev-tools. This file should not have to be modified, but can be used to extend your development needs. Upon building, `webpack` will step in and create a bundle with `src/main/index.js` as its entry file.

File Tree

During development

Note: Some files/folders may differ based on the settings chosen during `vue-cli` scaffolding.

```
my-project
├─ .electron-vue
│  └─ <build/development>.js files
├─ build
│  └─ icons/
├─ dist
│  ├─ electron/
│  └─ web/
├─ node_modules/
├─ src
│  ├─ main
│  │  ├─ index.dev.js
│  │  └─ index.js
│  ├─ renderer
│  │  ├─ components/
│  │  ├─ router/
│  │  ├─ store/
│  │  ├─ App.vue
│  │  └─ main.js
│  └─ index.ejs
├─ static/
├─ test
│  ├─ e2e
│  │  ├─ specs/
│  │  ├─ index.js
│  │  └─ utils.js
│  ├─ unit
│  │  ├─ specs/
│  │  ├─ index.js
│  │  └─ karma.config.js
│  └─ .eslintrc
├─ .babelrc
├─ .eslintignore
├─ .eslintrc.js
├─ .gitignore
├─ package.json
└─ README.md
```

Production builds

```
app.asar
├─ dist
│  └─ electron
│     ├─ static/
│     ├─ index.html
│     ├─ main.js
│     └─ renderer.js
├─ node_modules/
└─ package.json
```

As you can probably tell, almost everything is stripped down in final production builds. This is almost mandatory when distributing electron apps, as you do not want your users to download bloated software with a large file size.

Renderer Process

> Since Electron uses Chromium for displaying web pages, Chromium's multi-process architecture is also used. Each web page in Electron runs in its own process, which is called the renderer process.
>
> In normal browsers, web pages usually run in a sandboxed environment and are not allowed access to native resources. Electron users, however, have the power to use Node.js APIs in web pages allowing lower level operating system interactions.

From the Electron Documentation

On the subject of `vue` and `vuex`

vue components

If you are unfamiliar with Vue components, please give this a read. Using components gives our large complex applications more organization. Each component has the ability to encapsulate its own CSS, template, and JavaScript functionality.

Components are stored in `src/renderer/components`. When creating child components, a common organization practice is to place them inside a new folder with the name of its parent component. This is especially useful when coordinating different routes.

```
src/renderer/components
├─ ParentComponentA
|   ├─ ChildComponentA.vue
|   └─ ChildComponentB.vue
└─ ParentComponentA.vue
```

vue routes

For more information about `vue-router` click here. In short, it is encouraged to use `vue-router` as creating a Single Page Application is much more practical when making Electron applications. Do you really want to manage a bunch of BrowserWindows and then communicating information between everything? Probably not.

Routes are held in `src/renderer/router/index.js` and defined like so...

```
{
  path: '<routePath>',
  name: '<routeName>',
  component: require('@/components/<routeName>View')
}
```

...where both `<routePath>` and `<routeName>` are variables. These routes are then attached to the component tree using the `<router-view>` directive in `src/renderer/App.vue`.

Notice

When using `vue-router`, do not use **HTML5 History Mode**. This mode is strictly meant for serving files over the `http` protocol and does not work properly with the `file` protocol that Electron serves files with in production builds. The default `hash` mode is just what we need.

vuex modules

Describing `vuex` is not the easiest thing to do, so please read this for a better understanding of what problem it tries to solve and how it works.

electron-vue takes advantage of `vuex`'s module structure to create multiple data stores, which are saved in `src/renderer/store/modules`.

Having multiple data stores can be great for organization, but it can also be annoying to have to import each and every one. But don't fret, as `src/renderer/store/modules/index.js` does the dirty work for us! This little script lets `src/renderer/store/index.js` import all of our modules in a one-shot manner. If all that didn't make sense, just know you can easily duplicate the given `Counter.js` module and it will be loaded in "magically".

Main Process

> In Electron, the process that runs package.json's main script is called the main process. The script that runs in the main process can display a GUI by creating web pages.

From the Electron Documentation

Since the `main` process is essentially a full node environment, there is no initial project structure other than two files.

`src/main/index.js`

This file is your application's main file, the file in which `electron` boots with. It is also used as `webpack`'s entry file for production. All `main` process work should start from here.

`app/src/main/index.dev.js`

This file is used specifically and only for development as it installs `electron-debug` & `vue-devtools`. There shouldn't be any need to modify this file, but it can be used to extend your development needs.

On the subject of using `__dirname` & `__filename`

Since the `main` process is bundled using `webpack`, the use of `__dirname` & `__filename` **will not** provide an expected value in production. Referring to the File Tree, you'll notice that in production the `main.js` is placed inside the `dist/electron` folder. Based on this knowledge, use `__dirname` & `__filename` accordingly.

If you are in need of a path to your `static/` assets directory, make sure to read up on Using Static Assets to learn about the super handy `__static` variable.

```
app.asar
├─ dist
│  └─ electron
│     ├─ static/
│     ├─ index.html
│     ├─ main.js
│     └─ renderer.js
├─ node_modules/
└─ package.json
```

Webpack Configurations

electron-vue comes packed with three separate webpack config files located in the `.electron-vue/` directory. Aside for the optional use of the `web` output, both `main` and `renderer` are similar in setup. Both make use of `babel-preset-env` to target `node@7` features, use `babili`, and treat all modules as `externals`.

`.electron-vue/webpack.main.config.js`

Targets electron's `main` process. This configuration is rather bare, but does include some common `webpack` practices.

`.electron-vue/webpack.renderer.config.js`

Targets electron's `renderer` process. This configuration handles your Vue application, so it includes `vue-loader` and many other configurations that are available in the official `vuejs-templates/webpack` boilerplate.

White-listing Externals

One important thing to consider about this config is that you can whitelist specific modules to not treat as webpack `externals`. There aren't many use cases where this functionality is needed, but for the case of Vue UI libraries that provide raw `*.vue` components they will need to be whitelisted, so `vue-loader` is able to compile them. Another use case would be using webpack `alias`es, such as setting `vue` to import the full Compiler + Runtime build. Because of this, `vue` is already in the whitelist.

`.electron-vue/webpack.web.config.js`

Targets building your `renderer` process source code for the browser. This config is provided as a strong starting base if you are in need of publishing to web. **electron-vue does not support web output further than what is provided.** Issues related to web output will most likely be deferred or closed.

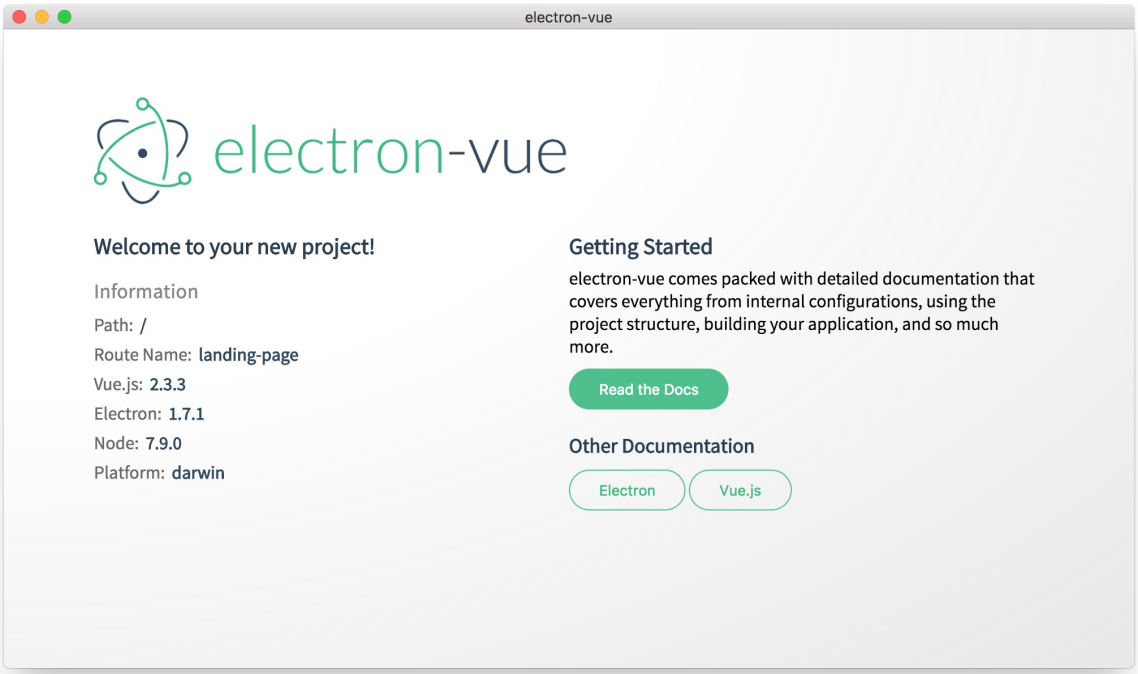
Development

Starting the development setup

After you have installed dependencies with `yarn` or `npm install`, then run...

```
yarn run dev # or npm run dev
```

...and boom! You now have a running electron-vue app.



This boilerplate comes with a few landing-page components that are easily removable.

Entry `index.html`

electron-vue makes use of `html-webpack-plugin` to create the `index.html` in production builds. During development you will find a `index.ejs` in the `src/` directory. It is here where you can make changes to your entry HTML file.

If you are unfamiliar with how this plugin works, then I'd encourage you take a look at its documentation. But in short, this plugin will automatically inject production assets including `renderer.js` and `styles.css` into a final minified `index.html`.

`index.ejs` during development

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title><%= htmlWebpackPlugin.options.title %></title>
    <%= ... %>
  </head>
  <body>
    <div id="app"></div>
    <!-- webpack builds are automatically injected -->
  </body>
</html>
```

`index.html` in production (non-minified)

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>app</title>
    <link href="styles.css" rel="stylesheet">
  </head>
  <body>
    <div id="app"></div>
    <script type="text/javascript" src="renderer.js"></script>
  </body>
</html>
```

On the subject of using CDNs

Although the benefits of using assets served from a CDN can be great for your application's final build size, I would advised against using them. The main reason being is that by doing so you are assuming the application always has access to the internet, which is not always the case for Electron apps. This becomes a rather major issue with CSS frameworks like bootstrap, as your app will quickly become an un-styled mess.

> "I don't care, I still want to use a CDN"

If you are determined to still use a CDN, then you can still do so by adding the tags to your `src/index.ejs` file. Just make sure to set up proper UI/UX flows for when you app is offline.

Vue Plugins

electron-vue comes packed with the following `vue` plugins that can be installed during `vue-cli` scaffolding...

- axios (web requests)
- vue-electron (attach electron APIs to Vue object)
- vue-router (single page application routes)
- vuex (flux-inspired application architecture)

`axios`

> Promise based HTTP client for the browser and node.js

If you are familiar with `vue-resource`, then `axios` will feel very familiar as most of the API is nearly identical. You can easily import `axios` in your `main` process scripts or use with `this.$http` & `Vue.http` in the `renderer` process. Please note that during development you may run into issues with CORS since requests are passed through `webpack-dev-server`. As a small workaround, you can disable `webSecurity` within the BrowserWindow configuration, but please do remember to only disable this during development. Disabling this in production is highly not recommended and can create a serious security risk for your final application!

`vue-electron`

> The vue plugin that attaches electron APIs to the Vue object, making them accessible to all components.

A simple `vue` plugin that makes electron APIs easily accessible with `this.$electron`, no longer needing to import `electron` into every component necessary.

`vue-router`

> `vue-router` is the official router for Vue.js. It deeply integrates with Vue.js core to make building Single Page Applications with Vue.js a breeze.

The provided project structure should feel familiar to the setup provided in the official `vuejs-templates/webpack` boilerplate.

`vuex`

> Vuex is a **state management pattern + library** for Vue.js applications. It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion.

The provided project structure is rather bare but does encourage the use of `vuex`'s module pattern to help organize your data stores. The extra `@/store/modules/index.js` let's your `vuex` store import all modules in a one-shot manner.

NPM Scripts

To help eliminate redundant tasks around the development process, please take note of some of the NPM scripts available to you. The following commands should be ran from your project's root directory. And of course, you can run any of the below commands using `yarn run <command>`.

`npm run build`

Build your app for production and package. More info can be found in the **Building Your App** section.

`npm run dev`

Run app in development.

You can also pass command line paramaters to the application with:

```
npm run dev --arg1=val1 --arg2
```

`npm run lint`

Lint all your `src/`'s and `test/`'s JS & Vue component files.

`npm run lint:fix`

Lint all your `src/`'s and `test/`'s JS & Vue component files and attempt to fix issues.

`npm run pack`

Run both `npm run pack:main` & `npm run pack:renderer`. Although these commands are available, there are not many cases where you will need to manually do this as `npm run build` will handle this step.

`npm run pack:main`

Run webpack to bundle `main` process source code.

`npm run pack:renderer`

Run webpack to bundle `renderer` process source code.

`npm run unit`

Run unit tests with Karma + Jasmine. More information on **Unit Testing**.

`npm run e2e`

Run end-to-end tests with Spectron + Mocha. More information on **End-to-end Testing**.

`npm test`

Runs both `npm run unit` & `npm run e2e`. More information on **Testing**.

Using CSS Frameworks

Although this may seem like a no brainer, I'd suggest you import your third-party CSS libraries into webpack using the `style-loader`, which is already setup for you.

Use Case

Say you want to use bootstrap, bulma, or materialize for your application. Go ahead and install your library from `npm` like you normally would, but instead of attaching the asset to `index.ejs` we will import the CSS in our JavaScript, specifically in `src/renderer/main.js`.

Example

Let's install `bulma`

```
npm install bulma --save
```

Then inside `src/renderer/main.js` let's add this line.

```
import 'bulma/css/bulma.css'
```

Alternatively, you can also include `bulma` from inside a component file.

App.vue

```
<style>
  @import "~bulma/css/bulma.css";
</style>
```

Now `webpack` will know to load in `bulma` for our application and make it available in our production builds.

Using Pre-Processors

One of the great benefits of using `vue-loader` with `webpack` is the ability to pre-process your HTML/CSS/JS directly in your Vue components files without much effort at all. For more information about this check **here**.

Use Case

Let's say we need to use Sass/SCSS for pre-processing our CSS. First, we need to install the proper `webpack` loader to handle this syntax.

Installing `sass-loader`

```
npm install --save-dev sass-loader node-sass
```

Once the loader we need is installed, everything is pretty much finished. `vue-loader` will magically take care of the rest. Now we can easily add `lang="sass"` or `lang="scss"` to our Vue component files. Notice we also installed `node-sass` as it is a dependent package for `sass-loader`.

Applying the `lang` attribute

So...

```
<style>
  body {
    /* CSS */
  }
</style>
```

...now becomes...

```
<style lang="scss">
  body {
    /* SCSS */
  }
</style>
```

The same principles apply for just about any other pre-processor. So maybe you need coffeescript for your JS? Simply install the coffeescript-loader and apply the `lang="coffeescript"` attribute to your `<script>` tag.

For more advanced use of this feature please head over to the vue-loader documentation for more information.

Using Sass/SCSS globals

When using Sass/SCSS for your CSS syntax, it's very beneficial to make use of global variables/mixins throughout all Vue component files. Here's how to make that happen.

Use Case

This example demonstrates how to apply a `globals.scss` to all Vue component files. This documentation assumes you have already setup `sass-loader` in your development environment as mentioned above.

Define your globals

src/renderer/globals.scss

```
$brand-primary: blue;
$brand-accent: turquoise;
```

Inject `globals.scss` directly into `node-sass`

Edit the `vue-loader` config in **.electron-vue/webpack.renderer.config.js**

```
loaders: {
  sass: 'vue-style-loader!css-loader!sass-loader?indentedSyntax=1&data=@import "./src/renderer/globals"',
  scss: 'vue-style-loader!css-loader!sass-loader?data=@import "./src/renderer/globals";'
}
```

Use your globals

SomeComponent.vue

```
<style lang="scss">
  body { color: $brand-primary; }
</style>
```

Using Static Assets

If you have used the official `vuejs-templates/webpack` boilerplate before, then you should be familiar with the `static/` directory. It is here where you can place static assets that both the `main` and `renderer` process can consume. Using these assets within your Vue application is simple, but usage with `fs` and other modules that need a full path can be a little tricky. Thankfully, electron-vue provides a `__static` variable that yields the path to the `static/` directory in both development and production.

Use Case within `src` tags in Vue Components

Let's say I have a component that loads an image, but the image's path isn't known until some other task is completed. To keep things simple, let's just use a `data` variable to bind our `<img>`'s src.

SomeComponent.vue

```
<template>
  <img v-bind:src="imageUrl">
</template>

<script>
  export default {
    data () {
      // notice the url starts with `static/`
      return { imageUrl: 'static/imgs/unsplash.png' }
    }
  }
</script>
```

Here `webpack` will not bundle the `unsplash.png` image and the application will look inside the `static/imgs/unsplash.png` directory for the asset. Thanks to `vue-loader`, all of the dirty work is done for us.

Use Case within JS with `fs`, `path` and `__static`

Let's say we have a static asset that we need to read into our application using `fs`, but how do we get a reliable path, in both development and production, to the `static/` directory? electron-vue provides a global variable named `__static` that will yield a proper path to the `static/` directory. Here's how we can use it to read a simple text file in both development and production.

static/someFile.txt

```
foobar
```

SomeFile.js (`main` or `renderer` process)

```
import fs from 'fs'
import path from 'path'

let fileContents = fs.readFileSync(path.join(__static, '/someFile.txt'), 'utf8')

console.log(fileContents)
// => "foobar"
```

Please note that in production all files are packed with `asar` by default as it is highly recommended. Because of this, assets within the `static/` folder can only be accessed within `electron` since it is aware of this behavior. So if you are planning to distribute files to your users, that can for example open in a external program, you would first need to copy

those assets from your application into the user's document space or desktop. From there you could use the `shell.openItem()` electron API to open those assets.

An alternative method to this situation would be to configure `electron-packager` / `electron-builder` to set specific files to "unpack" from the `asar` archive in production. electron-vue has no plans to support this method; any issues related to this or how to set this up will be closed.

Read & Write Local Files

One great benefit of using `electron` is the ability to access the user's file system. This enables you to read and write files on the local system. To help avoid Chromium restrictions and writing to your application's internal files, make sure to take use of `electron`'s APIs, specifically the `app.getPath(name)` function. This helper method can get you file paths to system directories such as the user's desktop, system temporary files, etc.

Use Case

Let's say we want to have a local database store for our application. In this example we'll demonstrate with `nedb`.

```
yarn add nedb # or npm install nedb --save
```

src/renderer/datastore.js

Here we setup NeDB and point it to our `userData` directory. This space is reserved specifically for our application, so we can have confidence other programs or other user interactions should not tamper with this file space. From here we can import `datastore.js` in our `renderer` process and consume it.

```
import Datastore from 'nedb'
import path from 'path'
import { remote } from 'electron'

export default new Datastore({
  autoload: true,
  filename: path.join(remote.app.getPath('userData'), '/data.db')
})
```

src/renderer/main.js

To take this a step further, we can then import our datastore into `src/renderer/main.js` and attach it to the Vue prototype. Doing so, we are now able to access the datastore API through the usage of `this.$db` in all component files.

```
import db from './datastore'

/* Other Code */

Vue.prototype.$db = db
```

Debugging

Main Process

When running your application in development you may have noticed a message from the `main` process mentioning a remote debugger. Ever since the release of `electron@^1.7.2`, remote debugging over the Inspect API was introduced and can be easily accessed by opening the provided link with Google Chrome or through another debugger that can remotely attach to the process using the default port of 5858, such as Visual Studio Code.

```
┌ Electron -------------------

  Debugger listening on port 5858.
  Warning: This is an experimental feature and could change at any time.
  To start debugging, open the following URL in Chrome:
      chrome-devtools://devtools/bundled/inspector.html?experiments=true&v8only=true&ws=127.0.0.1:5858/22271e96-df65-
4bab-9207-da8c71117641

└ ----------------------------
```

Production Builds

Notice

Although it is possible to debug your application in production, please do know that production code is minified and highly unreadable compared to what you find during development.

`renderer` Process

There isn't much of a big difference here than it is in development. You can simply invoke the dev tools using the `BrowserWindow` APIs. Using the initial electron-vue setup, you can add the following snippet of code inside `src/main/index.js`, just after the `new BrowserWindow` construction, to force the dev tools to open on launch.

```
mainWindow.webContents.openDevTools()
```

`main` Process

Similar to what is mentioned above, you can also attach an external debugger to the `main` process to remotely debug your application. In order to activate the debugger in production you can add the follow snippet after the `app` import inside `src/main/index.js`. Then you can navigate Google Chrome to `chrome://inspect` and get connected.

```
app.commandLine.appendSwitch('inspect', '5858')
```

Building Your App

electron-vue supports both electron-packager and electron-builder to build and distribute your production ready application. Both build tools are backed by the amazing @electron-userland community and each have detailed documentation. During `vue-cli` scaffolding you will be asked which builder you will want to use.

`electron-packager`

If you are new to making electron applications or just need to create simple executables, then `electron-packager` is perfect for your needs.

`electron-builder`

If you are looking for full installers, auto-update support, CI building with Travis CI & AppVeyor, or automation of rebuilding native node modules, then `electron-builder` is what you will need.

Using `electron-packager`

All builds produced by `electron-packager` can be found within the `build` folder.

Building for all platforms

Please know that not all Operating Systems can build for all other platforms.

```
npm run build
```

Building for a specific platform

Platforms include `darwin`, `mas`, `linux` and `win32`.

```
# build for darwin (macOS)
npm run build:darwin
```

Cleaning

Delete all builds from `build`.

```
npm run build:clean
```

A note for non-Windows users

If you are wanting to build for Windows **with a custom icon** using a non-Windows platform, you must have wine installed. More Info.

Default building configurations

Further customization can be made at `.electron-vue/build.config.js` in accordance to `electron-packager`'s options found here. The name applied to your built application is set with the `productName` value in your `package.json`.

```
{
    // Target 'x64' architecture
    arch: 'x64',

    // Compress app using 'electron/asar'
    asar: true,

    // Directory of the app
    dir: path.join(__dirname, '../'),

    // Set electron app icon
    // File extensions are added based on platform
    //
    // If building for Linux, please read
    // https://github.com/electron-userland/electron-packager/blob/master/docs/api.md#icon
    icon: path.join(__dirname, '../build/icons/icon'),

    // Ignore files that would bloat final build size
    ignore: /(^\/(src|test|\.[a-z]+|README|yarn|static|dist\/web))|\.gitkeep/,

    // Save builds to `builds`
    out: path.join(__dirname, '../build'),

    // Overwrite existing builds
    overwrite: true,

    // Environment variable that sets platform
    platform: process.env.BUILD_TARGET || 'all'
}
```

Using `electron-builder`

All builds produced by `electron-builder` can be found within the `build` directory.

Building

```
npm run build
```

Building unpacked directory

Produce simple executable without full installer. Useful for quick testing.

```
npm run build:dir
```

Default building configuration

Further customization can be made at `package.json` in accordance to `electron-builders` 's options found here.

```
"build": {
  "productName": "ElectronVue",
  "appId": "org.simulatedgreg.electron-vue",
  "dmg": {
    "contents": [
      {
        "x": 410,
        "y": 150,
        "type": "link",
        "path": "/Applications"
      },
      {
        "x": 130,
        "y": 150,
        "type": "file"
      }
    ]
  },
  "directories": {
    "output": "build"
  },
  "files": [
    "dist/electron",
    "node_modules/",
    "package.json"
  ],
  "mac": {
    "icon": "build/icons/icon.icns"
  },
  "win": {
    "icon": "build/icons/icon.ico"
  },
  "linux": {
    "icon": "build/icons"
  }
}
```

Automated Deployments using CI

When using electron-vue's `electron-builder` configuration, you are also provided a `appveyor.yml` and `.travis.yml` for automated deployments. Both config files are setup for building your electron application and pushing artifacts to a GitHub release, Bintray, etc. Travis CI is used to build both `linux` and `darwin` (macOS) while AppVeyor is used to build `win32`. Both services are free for OSS projects.

Setting up Travis CI/AppVeyor

1. Create an account over at Travis CI / AppVeyor
2. Link your GitHub repository that has your electron-vue project
3. Visit https://github.com/settings/tokens and hit **Generate new token** (the same token can be used for both Travis CI & AppVeyor)
    i. Set a **Token description**
    ii. Check the **public_repo** scope
    iii. Hit **Generate token**
4. Copy your new token and save for later
5. Open your repository settings on Travis CI / AppVeyor to add a new **Environment Variable**
    i. Set the name of the variable to `GH_TOKEN`
    ii. Set the value of the variable to the GitHub access token you just created
    iii. **Save** the new variable and ensure encryption is enabled

At this point, everything should be setup. Travis CI/AppVeyor by default will watch for any pushes to your `master` branch. When a push is made, Travis CI/AppVeyor will then clone down your repository to its server and begin the build process. During the final stages, `electron-builder` with see the `GH_TOKEN` environment variable and create a draft release and upload the artifacts to your public GitHub repository. From this point, you can edit the draft release and then publish it to the world. After publishing your release, make sure future releases are marked with a new version number by updating your `package.json`.

Auto Updating

Enabling your application to receive automatic updates is a super nice feature to have, but know that **Code Signing** is required. You can setup code signing by adding a few more environment variables based on what `electron-builder` needs described here. Once you have your certificates setup, you can then install `electron-updater` and comment out the chunk of code at the bottom of `src/main/index.js` to enable auto updating.

If you are like most people and do not have a fancy code signing certificate, then you can always use the GitHub API to check for new releases. When a new release is detected, provide a notification within your application to point users to a download page where they can download and install the new build. Thanks to the amazing installer that `electron-builder` provides, user's do not have to uninstall the current version and the new installation will replace the old while still persisting any web storage or `userData` files.

Testing

electron-vue supports both unit testing and end-to-end testing for the `renderer` process and is heavily inspired by the testing setup provided with the official `vuejs-templates/webpack` boilerplate. During `vue-cli` scaffolding you will have the option to include testing support.

Unit testing

Run unit tests with Karma + Mocha

```
npm run unit
```

End-to-end testing

Run end-to-end tests with Spectron + Mocha

```
npm run e2e
```

Running all tests

```
npm test
```

On the subject of CI testing

If your decided to use `electron-builder` as your build tool when scaffolding, then you can easily test your application on both Travis CI & AppVeyor for `darwin`, `linux`, and `win32`. Inside both `.travis.yml` and `appveyor.yml` you will find commented sections you can quickly un-comment to enable testing. Make sure to read up on **Automated Deployments using CI** for further information.

Unit Testing

electron-vue makes use of the Karma test runner and the Mocha test framework (with Chai) for unit testing.

Both Mocha and Chai are integrated using `karma-mocha` and `karma-chai` respectively, so all APIs such as `expect` are globally available in test files.

Running Tests

```
# Begin Karma
npm run unit
```

File Structure

```
my-project
├─ test
|  ├─ unit
|  |  ├─ specs/
|  |  ├─ index.js
└─ └─ └─ karma.conf.js
```

For the most part, you can ignore both `index.js` and `karma.conf.js` and focus solely on writing `specs/` .

`specs/`

Inside this directory is where actual tests are written. Thanks to the power of webpack, you have full access to ES2015 and supported loaders.

`index.js`

This is the entry file used by `karma-webpack` . The purpose of this file is to gather all test and source code in a "one-shot" manner.

`karma.conf.js`

Here you will find the actual `karma` configuration, set up with spec/coverage reporters. Further customization can be made in accordance to the official karma documentation.

Mocking Dependencies

electron-vue comes with `inject-loader` installed by default. For usage with Vue component files see `vue-loader` docs on testing with mocks.

End-to-End Testing

electron-vue makes use of Spectron and the Mocha (with Chai) test framework for end-to-end testing. Mocha & Chai APIs, including `expect`, `should`, and `assert`, are made available in global scope.

Running tests

```
# Begin Mocha
npm run e2e
```

Note

Before running end-to-end tests, a `npm run pack` is called to create a production build that Spectron can consume during tests.

File Structure

```
my-project
├─ test
|  ├─ e2e
|  |  ├─ specs/
|  |  ├─ index.js
└─ └─ └─ utils.js
```

For the most part, you can ignore `index.js` and focus solely on writing `specs/`.

`specs/`

Inside this directory is where actual tests are written. Thanks to the power of `babel-register`, you have full access to ES2015.

`index.js`

This file acts as the main entry to Mocha and gathers all tests written in `specs/` for testing.

`utils.js`

Here you will find generic functions that could be of use throughout your `specs/`. Base functions include a `beforeEach` and `afterEach` that handle the electron creation/destruction process.

On the subject of Spectron

Spectron is the official electron testing framework that uses both ChromeDriver and WebDriverIO for manipulating DOM elements.

Using WebDriverIO

As stated in the Spectron documentation, access to WebDriverIO APIs can be accessed through `this.app.client`. Since electron-vue uses Mocha, the context of `this` is shared between `afterEach`, `beforeEach`, and `it`. Because of this, it is important to note that ES2015 arrow functions cannot not be used in certain situations as the context of `this` will be overwritten (more info).

Meta

Thank You

Wow, thank you guys so much for helping make electron-vue one of the top 3 `vue-cli` templates (that I can find) on GitHub. I never thought this project would ever take off like it has today. Thinking back, I originally made this boilerplate (in **May 2016**) for a personal closed sourced project and decided to open source (the boilerplate itself) when I knew I had a majority of the pieces together. Fast-forward to today and there have been so many new features implemented and amazing support from the community. I also want to give a special shoutout to those in the community helping answer issues when I'm not able to. You guys have absolutely no obligation to do anything but you do anyway, and I am grateful for that.

If you are reading this, then I can almost assume you really do enjoy electron-vue. A lot of time was spent creating this boilerplate. If you are feeling generous, then feel free to leave a tip if you want. Surely future development of electron-vue is not dependent upon donations, but its always an option if you decide to use it.

PayPal.me

FAQs

Why is my electron app blank after running `npm run dev` ?

TL;DR

Make sure you don't have a personal **proxy** setup that could tamper with `webpack-dev-server` .

Why does my electron app show a file explorer?

TL;DR

Your `src/renderer` contains error(s). Check console, fix errors, then refresh electron with `CommandOrControl+R` .

Long answer

If error(s) are present in you `src/renderer` this creates conflicts with ESLint on first run. In turn, an INVALID webpack `renderer.js` is produced which interrupts `HtmlWebpackPlugin` from creating `index.html` . Since `webpack-dev-server` doesn't have the `index.html` ready to serve, the server falls back to the file explorer.

Why is `vue-devtools` / `devtron` missing?

Make sure to close and reopen the developer tools panel on first launch if they are missing. Also check your terminal check for any error messages that may occur during installation.

Where do I put Static Assets?

Using Static Assets

Why did `npm run lint` end with an error?

The default nature of eslint is to print linting errors to console, and if there is any found the script will end with a non-zero exit (which produces npm errors). This is normal behavior.

Why can't I load my app in a web browser?

#195

How do import `jquery` ?

If you are wanting to use `bootstrap` , I'm going to have to stop you right there. Using both `vue` and `jquery` in the same environment is a bad practice and leads to the two frameworks colliding with each other. I would highly recommend using a `bootstrap` alternative that uses `vue` for its JavaScript functionality. Some recommendations include `bootstrap-vue` and `vue-strap` . For whatever reason you must use `jquery` , seek guidance from `webpack` 's documentation about the `ProvidePlugin` or see #192.

How can I debug the `main` process?

When using `electron@^1.7.2` you can open up Google Chrome, head to `chrome://inspect` , and then pop open the remote electron process while your application is running in development mode.

Electron Documentation

New Releases

electron-vue has evolved greatly since its initial creation in May of 2016 and has introduced many new fantastic features. Can you believe there was a time `vue-cli` scaffolding wasn't supported? Development of new features is not planned to end anytime soon. The only down side to new bells & whistles every now and then is having your project stuck on an older scaffold. Since electron-vue takes advantage of `vue-cli` there unfortunately isn't a structured way to *version* the boilerplate or make it *updatable*.

Major updates of electron-vue will be made through GitHub Milestones and will include many new features/bug fixes at a time, making these releases the optimal time to migrate any existing projects to a newer scaffold. These milestones are not usually planned, but arise as feature requests add up in the issue tracker.

Past Milestones

Multiplex

- Migration to `webpack` 2
- Support for `electron-builder`
- Support for `main` process bundling
- General bugfixing

Minimize

- Migration to single `package.json` structure
- Travis CI / AppVeyor configs for `electron-builder` users
- Minimal web output of `renderer` process
- Migration to `axios`
- Full support for `main` process bundling
- Rewrite of development and build scripts
- Migration to `babili` to remove the need of transpiling down completely to ES5
- Support for `static/` assets when using modules that require a full path (`__static`)

Migration Guide

The following documentation attempts to explain how migrating your project *should* be accomplished, but may not be a full proof method as overall project structure is always up for change.

1. Scaffold a fresh version of electron-vue using `vue init simulatedgreg/electron-vue my-project`
2. Copy over current project `src` files into the new scaffold's `src` directory
3. Copy over `package.json` dependencies from current project to the new scaffold's `package.json`
4. Install dependencies with `yarn` or `npm install`
5. Run project in development mode (`yarn run dev` or `npm run dev`)
6. Watch console for errors to fix

Just as previously mentioned above, there isn't a full proof method for migrating to a new scaffold, but these are typically going to be the major steps to get you nearly all the way there. Any personal modifications to project structure or handling of assets will be up to you or your team to migrate. Make sure to check out the rest of the documentation as it will always reflect the current version of electron-vue from the `master` branch.

Contributing

Wanting to help with this boilerplate? Feel free to submit a pull request. Before getting ready to submit anything, make sure to check out the following...

JavaScript Standard Style

To ensure all JS follows basic ***style standards*** make sure it follows these rules.