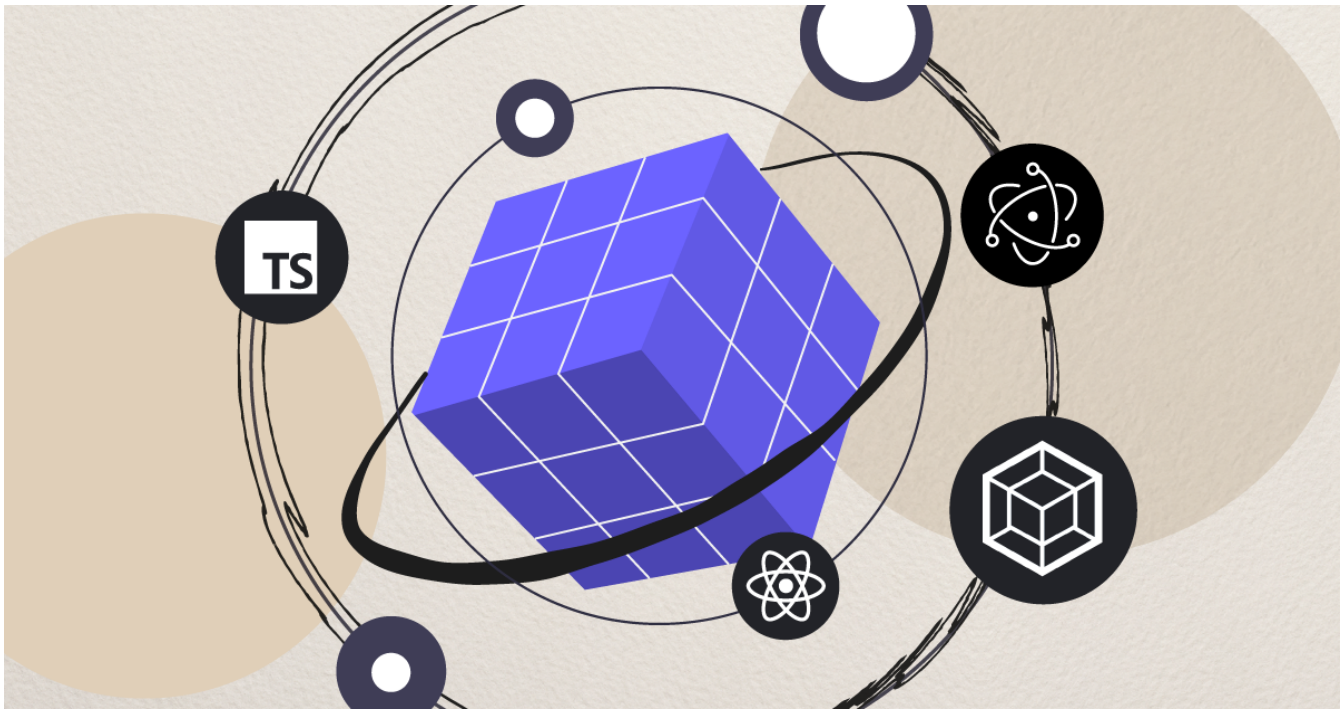


If you need to build a desktop application today, **Electron** is an increasingly common choice. It is cross-platform and is built using the same web technologies that you probably already know.

We're long-time users of Electron at SitePen, and have previously talked about **Setting up Electron with Dojo**. Here we will explore an opinionated approach to setting up Electron: TypeScript, React and Webpack.

We'll start with a basic Electron project and progressively build it into an enterprise-ready solution.

Initialize an empty Electron project

First we need a vanilla Electron project. It will be virtually identical to the official **Electron First App** tutorial and the **Electron Quickstart repository**.

Electron has two separate processes: a **main** process, which is Electron itself, and a **render** process, which is essentially a web page that Electron loads in a Chromium-based browser.

Install dependencies

```
npm init -y
npm install --save-dev electron
```

Electron (main) entry point

```
// src/electron.js
const { app, BrowserWindow } = require('electron');

function createWindow () {
  // Create the browser window.
  let win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    }
  });

  // and load the index.html of the app.
  win.loadFile('index.html');
}

app.on('ready', createWindow);
```

Electron (render) entry point

```
<!-- // src/index.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World!</title>
  </head>
  <body>
    <div id="app">
      <h1>Hello World!</h1>
    </div>
  </body>
</html>
```

We can run the app with `npx electron src/electron.js`. We'll add this in our `package.json` as a script.

```
// package.json
"scripts": {
  "start": "electron src/electron.js"
}
```

Adding TypeScript

The boilerplate JavaScript is also valid TypeScript, so let's rename `src/electron.js` to `electron.ts`. We just need to install the TypeScript compiler and configure it.

Install dependencies

```
npm install --save-dev typescript
```

TypeScript configuration

```
touch tsconfig.json
```

Update npm scripts

```
"scripts": {
  "build": "tsc src/electron.ts"
}
```

Adding Webpack

Next we'll set up **Webpack** to optimize our application. Webpack configuration consists of an array of **entry points**. Webpack processes each entry point by passing the file (and its dependencies) through a **loader**. Loaders are selected via **rules**, often with a loader per file extension. Finally, Webpack dumps the output to a specified location.

We'll create a single entry point for our electron main process, add a loader for all `*.ts` files to pass through the TypeScript compiler, and tell Webpack to dump the output alongside the source files.

Install dependencies

```
npm install --save-dev webpack webpack-cli ts-loader
```

Webpack configuration

```
// webpack.config.js
module.exports = [
  {
    mode: 'development',
    entry: './src/electron.ts',
    target: 'electron-main',
    module: {
      rules: [{
        test: /\.ts$/,
        include: /src/,
        use: [{ loader: 'ts-loader' }]
      }]
    },
    output: {
      path: __dirname + '/src',
      filename: 'electron.js'
    }
  }
];
```

Here's a breakdown of each piece of the configuration:

- `mode: develop` Development build (as opposed to production).
- `entry: './src/electron.ts` Location of the entry point
- `target: 'electron-main'` Specifies which environment to target; Webpack knows about the electron main process specifically.
- `test: /\.ts$/` Specifies that this rule should match all files that end with the `.ts` extension.
- `include: /src/` Specifies that all files within `src` should be considered for matching this rule.
- `use: [{ loader: 'ts-loader' }]` Specifies which loader(s) to use when this rule matches.
- `path: __dirname + '/src'` Directory where all output files will be placed.
- `filename: 'electron.js'` Primary output bundle filename.

Update npm scripts

```
// package.json
"scripts": {
  "build": "webpack --config ./webpack.config.js",
  "start": "npm run build && electron ./src/electron.js"
}
```

Adding React

The React render process does not need to know it's being used within an Electron context, so setting up React is similar to setting up a vanilla React project.

Install dependencies

```
npm install --save-dev react react-dom @types/react @types/react-dom
```

React entry point

```
// src/react.tsx
import * as React from 'react';
import * as ReactDOM from 'react-dom';

const Index = () => {
    return <div>Hello React!</div>;
};

ReactDOM.render(<Index />, document.getElementById('app'));
```

TypeScript configuration

Our render entry point is `.tsx` and not `.ts`. The TypeScript compiler has built-in support for TSX (The TypeScript equivalent of JSX), but we need to tell TypeScript how to handle our TSX resources. Not surprisingly, we're using the React TSX variety.

```
// tsconfig.json
{
  "compilerOptions": {
    "jsx": "react"
  }
}
```

Next, we'll create a new entry point in Webpack's configuration. Webpack will process our entry point (and its dependencies) and load the result into our `index.html` via the html-webpack-plugin.

Install dependencies

```
npm install --save-dev html-webpack-plugin
```

Webpack configuration

```
// webpack.config.js
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = [
  ...
  {
    mode: 'development',
    entry: './src/react.tsx',
    target: 'electron-renderer',
    devtool: 'source-map',
    module: { rules: [{
      test: /\.ts(x?)$/,
      include: /src/,
      use: [{ loader: 'ts-loader' }]
    }] },
    output: {
      path: __dirname + '/dist',
      filename: 'react.js'
    },
    plugins: [
      new HtmlWebpackPlugin({
        template: './src/index.html'
      })
    ]
  }
];
```

This configuration is similar to that of our main process, but there are some new items:

- `target: 'electron-renderer'` Specifies which environment to target; Webpack knows about the electron renderer process specifically.
- `plugins ...` Specifies any plugins used during the build process. Plugins differ from loaders in that plugins operate at the **bundle** level and can more deeply integrate with the build process via hooks. Loaders operate at the **file** level. The `HtmlWebpackPlugin` will automagically add a reference to the output bundle in the specified `template` file.

Since the output path for our renderer files is no longer the `src` directory, we've instructed Webpack to put our resources in a new `dist` directory. Let's do the same for the main process' configuration.

```
// webpack.config.js
...
  output: {
    path: __dirname + '/dist',
    filename: 'electron.js'
  }
```

With our output files now inside the `dist` directory, we need to update our npm scripts to match.

```
"scripts": {
  ...
  "start": "npm run build && electron ./dist/electron.js"
}
```

Conclusion

And that's it! As it turns out, Electron is well suited for running the major front-end frameworks and Webpack is well suited for packaging multiple things at once. The whole process just needed a little demystification.

Need help creating your next desktop application or determining if Electron is the right approach for you? **Contact us** to discuss how we can help!

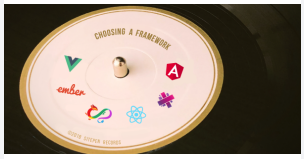
Follow SitePen for more articles just like this

Do you have any questions or want some expert assistance?

Let us know!

You might also enjoy

A Quick Look at Nest



FullStack London 2018: Choosing a

React Already Did That at All Things Open

Deploying a Dojo App with Docker

Framework **2018**