Wykrhm Reddy

Follow 33 Followers About

Creating standalone Desktop Applications with React, Electron and SQLite3

Wykrhm Reddy Oct 16, 2020 · 10 min read

There are quite a few tutorials on the Internet that cover the process of setting up React inside an Electron app but very few (if any) cover the solutions to common problems you run in to when packaging the app for distribution. In this blog, I am going to run you through how I setup a production ready work flow for creating a desktop app with **React** in **Electron** and **SQLite3** as a database packaged with the application.

In my most recent personal project, I built a Daily Build Updater for the 3D application Blender. This retrieves the latest daily build, compares it with your existing installation, below.

Let’s get started.

First and foremost, let us get React. The simplest and best way to do it would be using the **Create React App**. I will be using **yarn**. You can use **npm**.

```
yarn create react-app MyDesktopApp
cd MyDesktopApp
```

Now that we have React ready to go, let’s setup Electron. We’ll need to install a few packages for that.

- electron — *well .. Electron.*
- electron-builder — *To package the Electron app.*
- nodemon — *To monitor for changes during development and hot reload.*
- concurrently — *To launch the React app and Electron together*
- wait-on — *To wait for the React app to launch before the Electron app is launched.*

Dependencies:

- electron-is-dev — *Simple library to check if we're in development mode or not.*
- sqlite3 — *We'll use it later when we implement our database.*

```
yarn add electron electron-builder nodemon concurrently wait-on -D
yarn add electron-is-dev
```

We have the packages we need. We can get started with setting up Electron.

In your **public** folder, create a two new files called **electron.js** and **preload.js**. If you care to know, the reason we are creating it in this folder specifically is because when you build the React app, all contents of this folder get carried over to the *build* folder which makes things very simple for production. Additionally, any changes to files in this folder will trigger hot reload during development which can be very handy considering changes to the base Electron setup requires a full relaunch anyway.

The **electron.js** file will have the following code. It's basic Electron setup but I'll add comments where explanation is necessary.

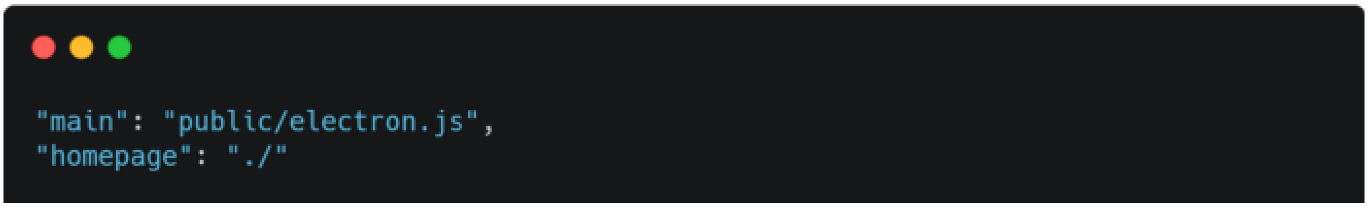
```
const { app, BrowserWindow } = require('electron'); // electron
const isDev = require('electron-is-dev'); // To check if electron is in development mode
const path = require('path');

let mainWindow;

```

```
    width: 600, // width of window
    height: 600, // height of window
    webPreferences: {
      // The preload file where we will perform our app communication
      preload: isDev
        ? path.join(app.getAppPath(), './public/preload.js') // Loading it from the publ
        : path.join(app.getAppPath(), './build/preload.js'), // Loading it from the buil
      worldSafeExecuteJavaScript: true, // If you're using Electron 12+, this should be
      contextIsolation: true, // Isolating context so our app is not exposed to random j
    },
  });

        // Loading a webpage inside the electron window we just created
  mainWindow.loadURL(
    isDev
      ? 'http://localhost:3000' // Loading localhost if dev mode
      : `file://${path.join(__dirname, '../build/index.html')}` // Loading build file if
  );

        // Setting Window Icon - Asset file needs to be in the public/images folder.
  mainWindow.setIcon(path.join(__dirname, 'images/appicon.ico'));

        // In development mode, if the window has loaded, then load the dev tools.
  if (isDev) {
    mainWindow.webContents.on('did-frame-finish-load', () => {
      mainWindow.webContents.openDevTools({ mode: 'detach' });
    });
  }
};

// ((OPTIONAL)) Setting the location for the userdata folder created by an Electron app.
app.setPath(
  'userData',
  isDev
    ? path.join(app.getAppPath(), 'userdata/') // In development it creates the userdata
    : path.join(process.resourcesPath, 'userdata/') // In production it creates userdata
);

// When the app is ready to load
app.whenReady().then(async () => {
  await createWindow(); // Create the mainWindow

```

```
    await session.defaultSession
      .loadExtension(
        path.join(__dirname, `../userdata/extensions/react-dev-tools`) // This folder sh
      )
      .then((name) => console.log('Dev Tools Loaded'))
      .catch((err) => console.log(err));
  }
});

// Exiting the app
app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') {
    app.quit();
  }
});

// Activating the app
app.on('activate', () => {
  if (mainWindow.getAllWindows().length === 0) {
    createWindow();
  }
});

// Logging any exceptions
process.on('uncaughtException', (error) => {
  console.log(`Exception: ${error}`);
  if (process.platform !== 'darwin') {
    app.quit();
  }
});
```

electron.js hosted with ❤ by GitHub view raw

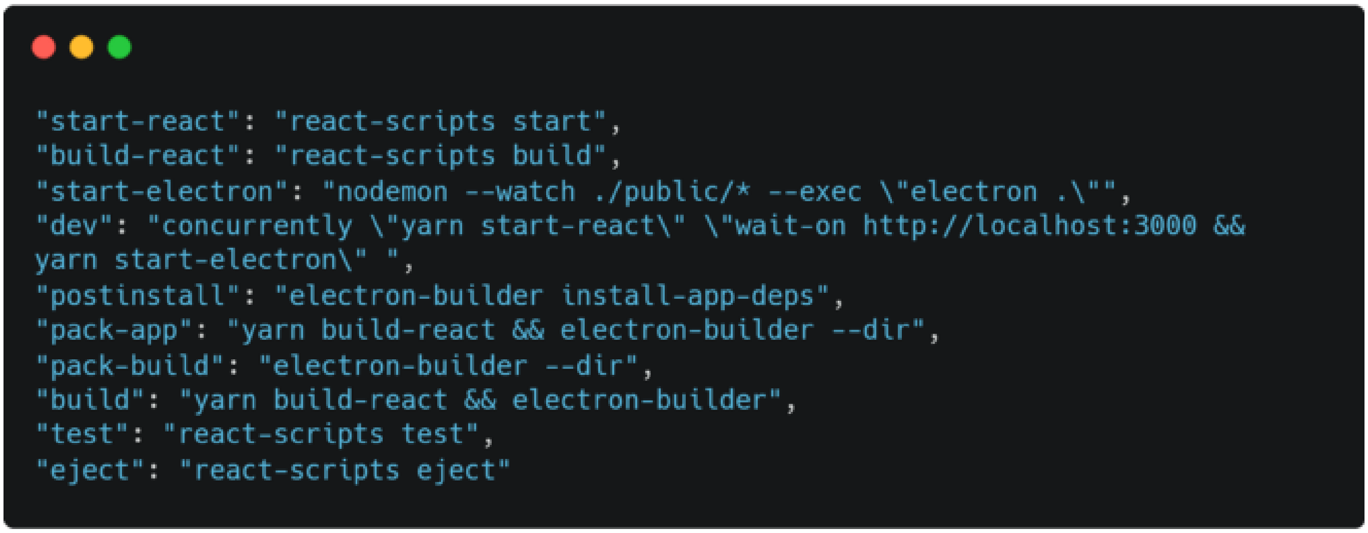
Now that we have defined the Electron window, let us test out the app by booting it in development mode. Go to **package.json** and add the following.



```
"main": "public/electron.js",
"homepage": "./"
```

and by setting homepage, you're letting React know to build the app keeping in mind that the base location is the default location — *This is required if your front is going to be a single page application*.

Now we need a few **scripts** in **package.json** to be changed and added. Rename "start" to "**start-react**" and build to "**build-react**" and then add the following. Your scripts should look something like this.



```
"start-react": "react-scripts start",
"build-react": "react-scripts build",
"start-electron": "nodemon --watch ./public/* --exec \"electron .\"",
"dev": "concurrently \"yarn start-react\" \"wait-on http://localhost:3000 &&
yarn start-electron\" ",
"postinstall": "electron-builder install-app-deps",
"pack-app": "yarn build-react && electron-builder --dir",
"pack-build": "electron-builder --dir",
"build": "yarn build-react && electron-builder",
"test": "react-scripts test",
"eject": "react-scripts eject"
```

To explain what they do —

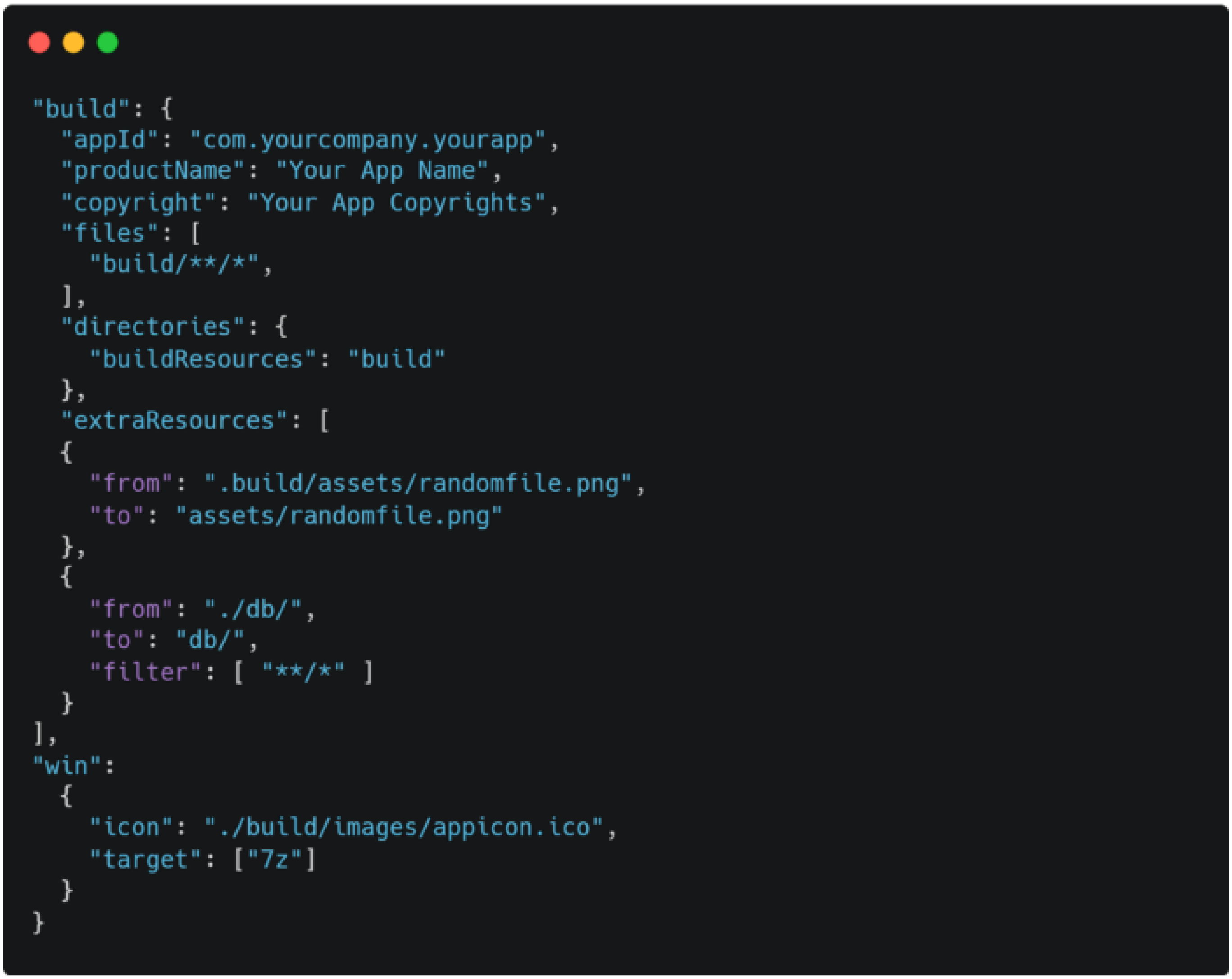
- start-react — *Will start just the React app only*
- build-react — *Will build the React app only*
- start-electron — *This will use nodemon to watch for changes in the public folder and then execute electron.* ***If you want to add more folders to be monitored, just add another*** `--watch` ***followed by the path to that folder***.
- dev — *Will first run React, wait for it to boot up and then start Electron.*
- postinstall — *It will make electron-builder install any dependencies we need for our app*
- pack-app — *Building the app can take time. Packing the app is shorter. It'll just pack the app so you can test your production builds.*

- test — *Comes with Create React App. Testing for your React app.*
- eject — *Comes with Create React App. Ejects your app from the CRA pipeline.*

Now just do `yarn dev` and it should boot up your React app inside an Electron Window.

Now that we have our development phase setup. Let us also set our build settings for production.

We need to add a **build** category to our **package.json** for **electron-builder** to know how we want our app built. There's **tons of options** but here's some basic ones for a Windows app. Refer to the link above for the full electron-builder documentation which is pretty clear.



```
"build": {
  "appId": "com.yourcompany.yourapp",
  "productName": "Your App Name",
  "copyright": "Your App Copyrights",
  "files": [
    "build/**/*",
  ],
  "directories": {
    "buildResources": "build"
  },
  "extraResources": [
  {
    "from": ".build/assets/randomfile.png",
    "to": "assets/randomfile.png"
  },
  {
    "from": "./db/",
    "to": "db/",
    "filter": [ "**/*" ]
  }
],
"win":
  {
    "icon": "./build/images/appicon.ico",
    "target": ["7z"]
  }
}
```

- build category — *Needed to provide information to electron-builder*
- appId — *Has to be unique so your app can be identifiable.*
- productName — *Name of your app*
- copyright — *Your app copyright info*
- files — *All the files you want to be packed in to the archives (asar) by electron-builder. **/* indicates every file inside a particular folder. In this case, I'm packing the entire* ***build*** *folder.*
- directories — *The location where electron-builder has to look for resources it needs. In our case, it is the build folder.*
- extraResources — *So here's the tricky part.*
  Along with the stuff you pack in the files category, you can pack extra files and folders with your application. These ***won't*** *be packed in the archive(asar) but rather get* ***copied*** *in the* ***resources*** *folder of the application.*
  So which files to add here?
  The general rule I follow is — if my app needs to edit or update info inside a particular file, then it needs to be an extra resource. If it doesn't need to, then it can go inside the file category above. In the above example, you can see a template on how to add individual files and also entire folders.
  If you are using ***worker_threads****, then you need to pack your worker thread file as an extraResource too.*
  If you are going to add a file as extraResources, then you can access that in your code with ***process.resourcesPath*** *which gives you the direct path to the resources folder. So in my case, if Iwant to access the db folder, it just type path.join(process.resourcesPath, "db/filename.db") for the production build.*
- win — *The win category contains instructions specific to a windows build. You add a category called mac if you want to do a mac and etc. Follow the electron-builder docs for info on this.*

tutorials online would recommend that you *enableNodeIntegration* on your Electron app and use the *ipcRenderer / ipcMain* directly to talk to your frontend.

But it is **NOT** safe to expose your Electron backend to the renderer. The same issue even persists with the usage of Electron **remote**. Electron themselves have recommended against this and the remote module has been deprecated in Electron 12 and will be removed in Electron 14.

So what is the solution? Well, that's where the **preload.js** which we created and the **contextIsolation: true** web preference we set earlier come in.

Firstly, the contextIsolation. By enabling this, you're isolating the Electron logic to run in a separate context to that of the front end you load. This will ensure that any rogue scripts do not have access to the all powerful Electron backend. But obviously, this would cut off the frontends access to the Electron backend.

So in order for these two sides to be able to talk to each other, we create a **bridge**. That might sound complicated but it really is not.

What you are doing is basically creating a simple API inside your preload.js by defining simple functions that are preloaded to the app. So the front end only has access to these functions only making your app secure.

```
const { ipcRenderer, contextBridge } = require('electron');

contextBridge.exposeInMainWorld('api', {
  // Invoke Methods
  testInvoke: (args) => ipcRenderer.invoke('test-invoke', args),
  // Send Methods
  testSend: (args) => ipcRenderer.send('test-send', args),
  // Receive Methods
  testReceive: (callback) => ipcRenderer.on('test-receive', (event, data) => { callback(
});
```

preload.js hosted with ❤️ by **GitHub** **view raw**

ipcRenderer will let us send signals which we will catch with ipcMain in electron.js.

contextBridge is basically creating a bridge between React and Electron with the already pre-defined functions that you've exposed to the front end. So React can only access these. Nothing else.

Keys used like 'test-invoke', 'test-send' and etc need to be **unique** so there's no clashes in the signals sent.

So instead of using ipcRenderer directly in React, you will now use **window.api.testInvoke('example argument)** which will trigger **ipcRenderer.invoke('test-invoke', 'example argument')** that you might be familiar with.

There's 3 basic ways of communication — I'll provide examples right after:

1. **invoke** — You send data from the frontend, process it with **ipcMain.handle** on the backend and return information to the frontend.
2. **send** — You send data from the frontend, process it in the backend with **ipcMain.on** and send back a reply when it is processed.
3. **on** — You receive data from the backend **event.sender** and process that with the help of a callback function.

*There is no functional difference between **invoke** and **send** as far as I know. invoke is a newer API and uses Promises. So I use invoke for Promise based operations and send for others. If there is something I am missing here, then please feel free to let me know.*

EXAMPLES:

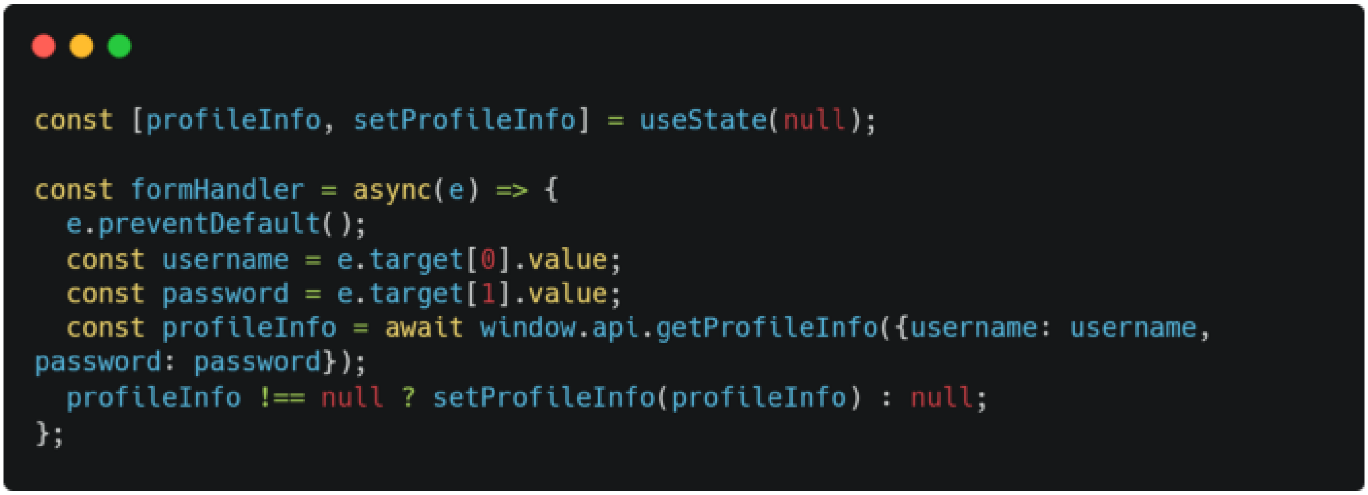
Let's say I want to enter username, password in React, process that info in Electron / Node and get back my profile information.

INVOKE

In preload.js

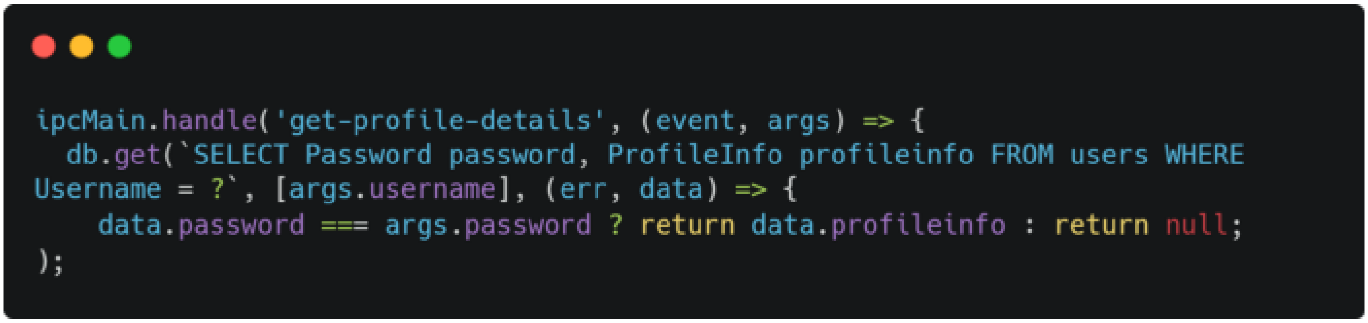
```
getProfileInfo: (args) => ipcRenderer.invoke('get-profile-details', args)
```

In ReactComponent.js



```
const [profileInfo, setProfileInfo] = useState(null);

const formHandler = async(e) => {
  e.preventDefault();
  const username = e.target[0].value;
  const password = e.target[1].value;
  const profileInfo = await window.api.getProfileInfo({username: username,
password: password});
  profileInfo !== null ? setProfileInfo(profileInfo) : null;
};
```

In electron.js

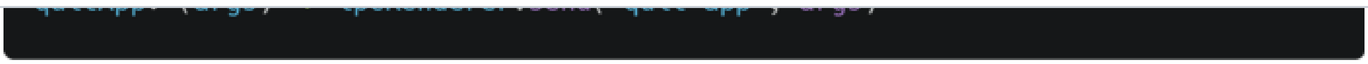


```
ipcMain.handle('get-profile-details', (event, args) => {
  db.get(`SELECT Password password, ProfileInfo profileinfo FROM users WHERE
Username = ?`, [args.username], (err, data) => {
    data.password === args.password ? return data.profileinfo : return null;
);
```

In the above example, when the user submits the form, the `formHandler` sends the form data via the `windows.api.getProfileInfo` . The `getProfileInfo` function we set in the preload will trigger the get-profile-details signal. The `get-profile-details` signal is caught in the electron.js file where the arguments are received, data is retrieved from the database, verified and then profile info is returned. This returned profile info is awaited for in the React app and if its not null, it is set to the React state.

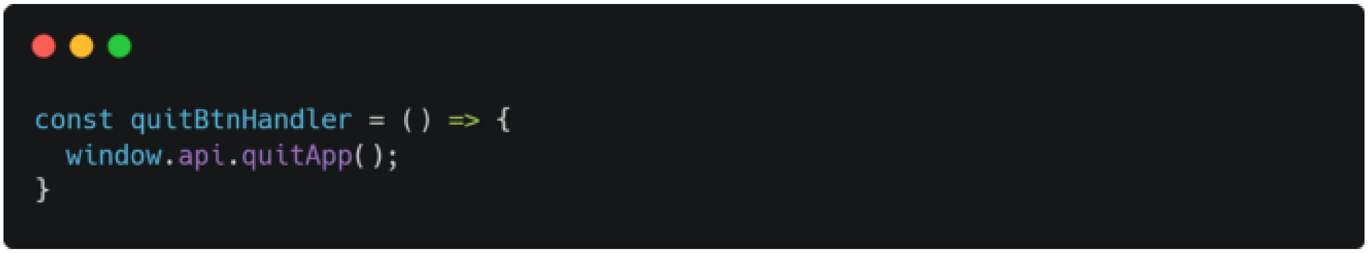
SEND

In preload.js



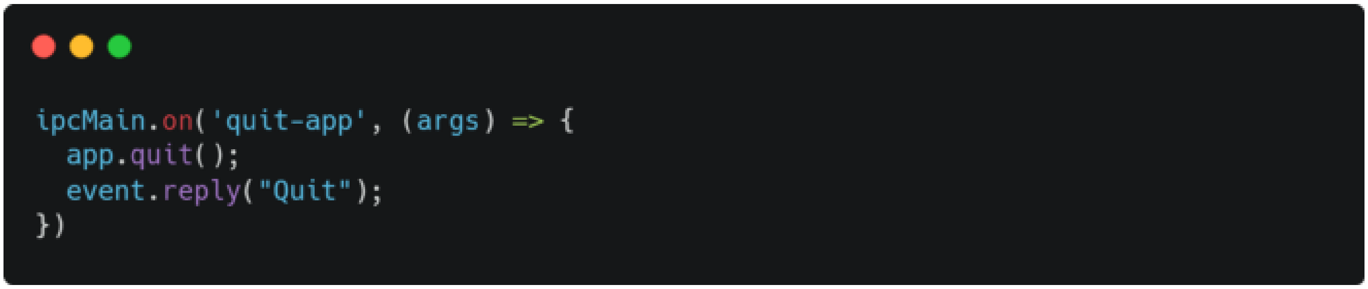
```
quitApp: (args) => ipcRenderer.send('quit-app', args)
```

In ReactComponent.js



```
const quitBtnHandler = () => {
  window.api.quitApp();
}
```

In electron.js

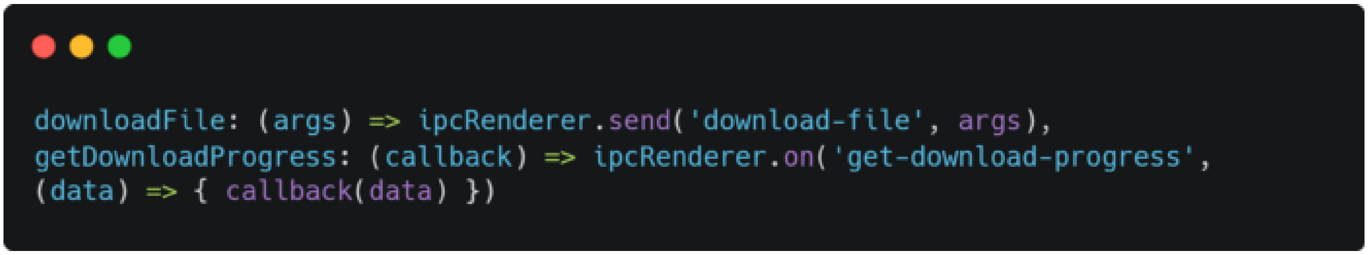


```
ipcMain.on('quit-app', (args) => {
  app.quit();
  event.reply("Quit");
})
```

Trigger quitApp() from the renderer. Quit the app using electron.js — Send a reply for funsies because who cares once the app is closed.

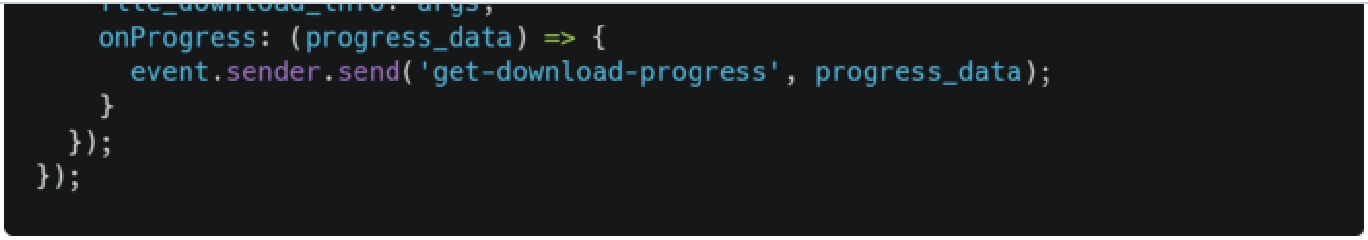
RECEIVE

In **preload.js**



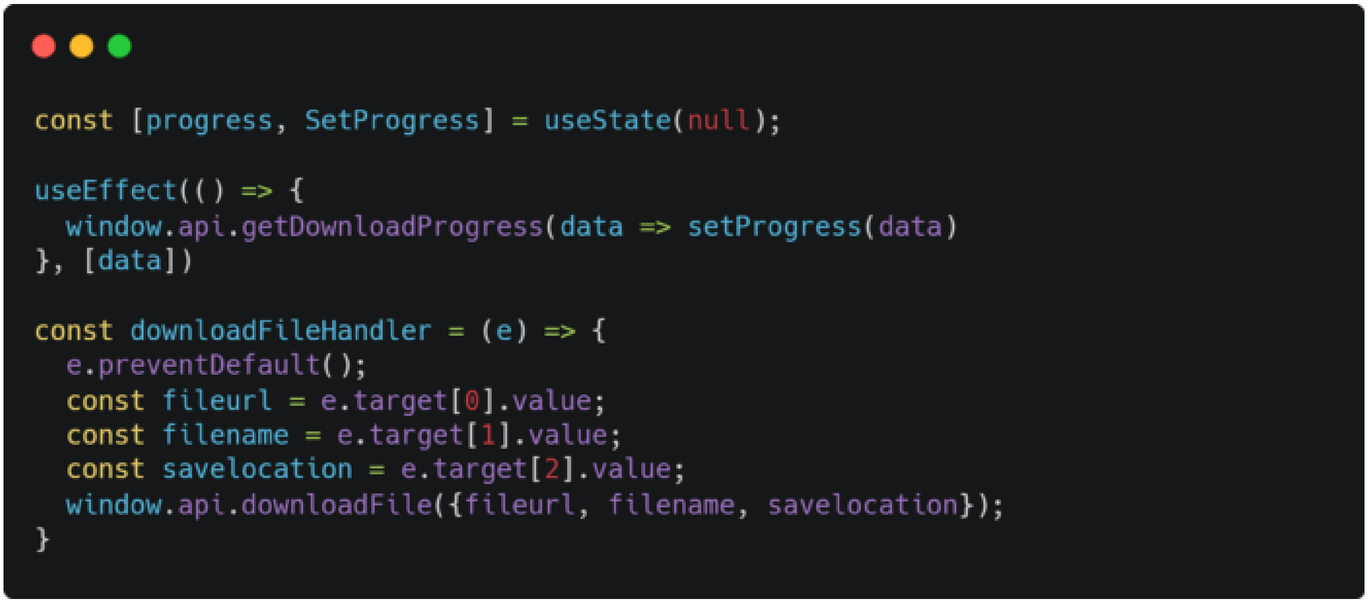
```
downloadFile: (args) => ipcRenderer.send('download-file', args),
getDownloadProgress: (callback) => ipcRenderer.on('get-download-progress',
(data) => { callback(data) })
```

In **electron.js**



```
      file_download_info: args;
      onProgress: (progress_data) => {
        event.sender.send('get-download-progress', progress_data);
      }
    });
  });
```

In **ReactComponent.js**



```
const [progress, SetProgress] = useState(null);

useEffect(() => {
  window.api.getDownloadProgress(data => setProgress(data)
}, [data])

const downloadFileHandler = (e) => {
  e.preventDefault();
  const fileurl = e.target[0].value;
  const filename = e.target[1].value;
  const savelocation = e.target[2].value;
  window.api.downloadFile({fileurl, filename, savelocation});
}
```

In this example, we click a button which triggers the **downloadFileHandler** which triggers the **downloadFile** signal passing the data from the form.

The **downloadFile** signal sent from **preload.js** with the data is caught by the electron.js file where the data is sent to an imaginary module called '*filedownloader*' whose function '*DownloadFile*' has a callback function for the progress called *onProgress*.

onProgress send back the progress info through an event sender called '**get-download-progress**' which is caught by **getDownloadProgress** in **preload.js.** The ReactComponent is constantly listening to getDownloadProgress with **useEffect** and updating the progress data each time it updates to the state.

Adding a Database

Use DB Browser for SQLite to create a new database. You can store it anywhere. I'd recommend creating a folder called **db** in your root folder.

NOTE: Make sure you add this folder to the extraResources path in the build category. Because we want to ship our database file along with our application.

We'll be using Node.js' `sqlite3` to communicate with our database.

```
yarn add sqlite3
```

sqlite3 generally needs bindings to be rebuilt. Because we have **postinstall** for electron-builder already setup, installing it right now should have fixed that issue for us. But if it did not, then just run `yarn postinstall` and watch for the success message.

```
const sqlite3 = require('sqlite3');

// Initializing a new database
const db = new sqlite3.Database(
  isDev
    ? path.join(__dirname, '../db/prefs.db') // my root folder if in dev mode
    : path.join(process.resourcesPath, 'db/prefs.db'), // the resources path if in produ
  (err) => {
    if (err) {
      console.log(`Database Error: ${err}`);
    } else {
      console.log('Database Loaded');
    }
  }
);
```

database.js hosted with ❤ by **GitHub** **view raw**

You can refer to the **SQLite documentation** on how to use it. It's pretty straight forward.

Conclusion

Let me know if I buggered up anything anywhere and feel free to buzz me up for any queries. You can catch me on Twitter at @wykrhm

React Reactjs Electron Sqlite Blender