

Electron js

Electron is a framework for building desktop applications using JavaScript, HTML, and CSS.

By embedding [**Chromium**](https://www.chromium.org/)and [**Node.js**](https://nodejs.org/)into its binary

Electron allows you to maintain one JavaScript codebase and create cross-platform apps that work on Windows, macOS, and Linux

### Advantages of Electron

 Cross-platform support (Windows, macOS, Linux)  Huge developer community

 Leverages web technologies

 Easy integration with Node.js modules

### Disadvantages of Electron

 High memory consumption  Larger app size

# Project Setup

## Install Node.js

If you havenʼt already:

 Download from <https://nodejs.org/>  Install the **LTS version**

## Create a New Project

create a new project using below commands

### Using Electron Forge

npx create-electron-app my-app-name

 Uses Electron Forge

 Preconfigured with sensible defaults  Great for quick prototyping

 Easy to build with

npm run make

### Folder Structure

my-app-name/

├── node\_modules/

├── out/ # Generated after build (`npm run make`)

├── src/

│ └── index.js

| └── index.html

| └── index.css

├── .gitignore

├── package.json

├── package.lock.json

└── forge.config.js

1. **Run project**

To run the project, ensure that you're inside the project folder :

cd my-app-name

On package.json ensure that scripts have start as electron-forge start as follows:

"scripts": {

"start": "electron-forge start",

"package": "electron-forge package",

.....

}

Then start the Electron app in development mode:

npm start

This will:

 Launch the Electron app

 Open the initial window ( )

src/index.js

### Build the Project

Once your Electron app is ready, you can **package** and **distribute** it.

### Step 1: Package the App

Use the following command:

npm run package

#### What this does:

 Packages your Electron app into a distributable format, **but not a final installer**

Output is placed in the directory by default.

out/

The contents include platform-specific builds of your app, like:

out/

├── my-app-name-darwin-x64/ # For macOS

├── my-app-name-win32-x64/ # For Windows

└── my-app-name-linux-x64/ # For Linux

### Step 2: Make (Create Installers)

To generate platform-specific **installers**, use:

npm run make

#### What this does:

 Creates OS-specific **installers** for your app (e.g., macOS).

for Windows,

.exe

for

.dmg

Uses Electron Forge's **makers** under the hood (e.g., Squirrel for Windows, DMG for macOS).

Output is placed in the same directory, inside a subfolder like:

out/

└── make/

├── dmg/

│ └── my-app-name-1.0.0.dmg

├── squirrel.windows/

│ ├── my-app-name-Setup-1.0.0.exe

└── zip/

# macOS installer

# Windows installer

└── my-app-name-1.0.0.zip # Linux or generic compressed

out/

# Overview of Main Concepts in Electron

Electron apps are built using **two types of processes** and **several key components**. Letʼs break it down.

## Main Process

 This is the **entry point** of your Electron app (usually ).  It runs in a **Node.js environment** and is responsible for:

src/index.js

 Creating and managing windows

 Handling application lifecycle events

 Interacting with the file system or OS APIs

 Think of it as the **“backendˮ** of your desktop app.

Complete window of chromium will be configured on my index.js The renderer page will be loaded from the main.js using the

mainWindow.loadFile("index.html");

We can also use Multiple windows as follows

let mainWindow new BrowserWindow({ width: 800,

height: 600

contextIsolation: true,

});

let secondWindow new BrowserWindow({ width: 400,

height: 300,

parent: mainWindow, modal: true,

show: false

});

=

=

If you also set , the child window will be modal — meaning it will **block**

**interaction** with the parent window until it is closed.

modal: true

If you also set , the child window will be modal — so that you can use

modal: false

either main window or second window at the same time .

Always set the property when creating a child window. This links the

parent

windows properly in terms of behavior and stacking (Z-order).

#### 1) Additional items that can be added in main.js

i) **Menus**

Create custom application menus (like File, Edit).

const  Menu } require('electron');

const template [

{

label: 'File', submenu: [

{ role: 'quit' } // Quit app

]

},

{

label: 'Help', submenu: [

{

label: 'About', click: () ⇒ {

console.log('About clicked');

}

}

]

}

];

const menu Menu.buildFromTemplate(template);

Menu.setApplicationMenu(menu);

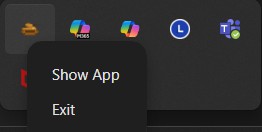


=

=

## Tray Icon

Add an icon in the system tray (bottom bar).



const  Tray, Menu } require('electron');

const path require('path');

let tray new Tray(path.join( dirname, 'icon.png'));

const contextMenu Menu.buildFromTemplate([

{ label: 'Show App', click: () ⇒ mainWindow.show() },

{ label: 'Quit', role: 'quit' }

]);

tray.setToolTip('My Electron App');

tray.setContextMenu(contextMenu);



=

=

=

## Global Shortcuts

Register keyboard shortcuts for the whole system.

const { globalShortcut } = require('electron'); app.whenReady().then(() ⇒ {

globalShortcut.register('CommandOrControl+X', () ⇒ { console.log('Shortcut triggered!');

});

});

## Auto-launch on Startup

Launch app automatically when system boots.

const AutoLaunch = require('auto-launch'); const myAppLauncher = new AutoLaunch({ name: 'MyElectronApp'

});

myAppLauncher.enable(); // You may want to check if it's already enabled

## App Events

Listen for lifecycle events (open, close, etc).

const { app } = require('electron');

app.on('ready', () ⇒ {

console.log('App is ready!');

});

app.on('window-all-closed', () ⇒ {

if (process.platform ! 'darwin') app.quit();

});

## IPC (Communication Between Windows)

Send/receive messages between **main** and **renderer** processes.

**Main Process:**

const { ipcMain } = require('electron');

ipcMain.on('message-from-renderer', (event, arg) ⇒ { console.log(arg); // do something

});

**Renderer (index.html or JS):**

const { ipcRenderer } = require('electron');

ipcRenderer.send('message-from-renderer', 'Hello Main!');

## Dialogs

Show native OS dialogs (open file, alert box, etc).

const { dialog } = require('electron');

dialog.showOpenDialog({ properties: ['openFile']

}).then(result ⇒ { console.log(result.filePaths);

});

## Notifications

Show system notifications.

new Notification({ title: 'Hello', body: 'This is a message.' }).show()

## Power Management

Prevent system sleep while app is running.

const { powerSaveBlocker } = require('electron');

const id = powerSaveBlocker.start('prevent-app-suspension');

# Preload.js

In Electron, the **renderer process** (your HTML/JS) is sandboxed from the **main process** for security. But sometimes, you need the renderer to **safely**

**communicate** with the main process or use limited Node.js features.  It **runs before the web page is loaded**

#### It has access to both Node.js and browser APIs

 It acts as a **safe bridge** between the main process and the renderer

## Context Isolation

makes sure that:

contextIsolation: true

 The **renderer JavaScript (your HTML)** runs in a separate **"isolated world"** from Electronʼs internal scripts

 This **prevents malicious code** from accessing Electron/Node.js internals

But your **can still expose safe APIs** to the renderer

preload.js

### Main Process (

**main.js**

Tell Electron to use the

**)**

file:

preload.js

const  BrowserWindow } = require('electron');

const path = require('path');

let mainWindow = new BrowserWindow({ width: 800,

height: 600, webPreferences: {

preload: path.join( dirname, 'preload.js'),

contextIsolation: true, // by default its true but ensure it by assigning it as tru

e

nodeIntegration: false

}

});

mainWindow.loadFile('index.html');

### Preload Script ( )

**preload.js**

This is where you **expose safe APIs** to your renderer.

safely exposes functions

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

// Expose a custom API to renderer

contextBridge.exposeInMainWorld('electronAPI', {

sendMessage: (msg) ⇒ ipcRenderer.send('toMain', msg),

onMessage: (callback) ⇒ ipcRenderer.on('fromMain', (event, data) ⇒ callback (data))

});

contextBridge.exposeInMainWorld(...)

This avoids exposing all Node.js APIs directly (which is insecure)

### Renderer ( or JS in your HTML)

**index.html**

Use the exposed API in your web page:

<script>

// Call function defined in preload.js

window.electronAPI.sendMessage('Hello from Renderer!');

// Listen for messages from main

window.electronAPI.onMessage((data) ⇒ { console.log('Received from main:', data);

});

</script>

# Package.json

It tells Electron (and npm) how to run your app and what it needs to work

### Example: Basic for Electron

**package.json**

{

"name": "crudapp", "productName": "crudapp", "version": "1.0.0",

"description": "My Electron application description", // Meta data "main": "src/index.js", // main file or start up file to run

"scripts": { // npm scripts

"start": "electron-forge start",

"package": "electron-forge package", "make": "electron-forge make",

"publish": "electron-forge publish",

"lint": "echo \"No linting configured\""

},

"keywords": [],

"author": "Balaji M [<balajibalajimanivannan@gmail.com>"](mailto:balajibalajimanivannan@gmail.com), "license": "MIT",

"dependencies": { // dependencies used on the time of production "electron-compile": "^6.4.4",

"electron-forge": "^5.2.4", "electron-squirrel-startup": "^1.0.1"

}

"devDependencies": { // dependencies used on the time of devolepment "electron": "^36.5.0"

}

## Forge Config.js

This file **controls how your Electron app is built, packaged, and published** using Electron Forge.

It includes:

 **Packager options** (like enabling ASAR)

**Makers** (to create

.exe

,

,

installers)

**Plugins** (like auto-unpack natives or Electron fuses)

.deb

.zip

🧪 **Sample:**

**forge.config.js**

Hereʼs your code, organized and explained with comments:

js

CopyEdit

const  FusesPlugin } = require('@electron-forge/plugin-fuses');

const  FuseV1Options, FuseVersion } = require('@electron/fuses');

module.exports = {

// Config for electron-packager packagerConfig: {

asar: true, // Bundle your app source code into a single .asar file

},

rebuildConfig: {}, // Optional: use if you need to rebuild native modules

// Makers create distributable formats makers: [

{

name: '@electron-forge/maker-squirrel', // For Windows .exe installer config: {},

},

{

name: '@electron-forge/maker-zip', // macOS .zip package platforms: ['darwin'],

},

{

name: '@electron-forge/maker-deb', // Linux .deb package (Debian/Ubunt

u)

config: {},

},

{

name: '@electron-forge/maker-rpm', // Linux .rpm package (Red Hat/Fedor

a)

config: {},

},

],

// Plugins add extra build features plugins: [

{

name: '@electron-forge/plugin-auto-unpack-natives', config: {},

},

// Fuses: enable/disable Electron runtime features at build time new FusesPlugin({

version: FuseVersion.V1, // Use Fuses V1

// Turn OFF node integration in main process binary

FuseV1Options.RunAsNode]: false,

// Enable secure cookies

FuseV1Options.EnableCookieEncryption]: true,

// Block dangerous environment-based features

FuseV1Options.EnableNodeOptionsEnvironmentVariable]: false,

FuseV1Options.EnableNodeCliInspectArguments]: false,

// Ensure ASAR contents canʼt be tampered with

FuseV1Options.EnableEmbeddedAsarIntegrityValidation]: true,

// Force the app to load only from ASAR, not from loose files

FuseV1Options.OnlyLoadAppFromAsar]: true,

}),

],

};

**Electron Fuses** are low-level runtime flags that can be **burned into the app binary**

to:

Disable risky Node.js features (e.g., )

-inspect

Prevent runtime environment injection Force the use of ASAR (bundled files only) Improve security for production apps

The and the fuses you configure:

**@electron-forge/plugin-fuses**

Are **only applied when you build/make/package** your Electron app

#### Do not run or have any effect during or development mode

**npm start**

You can test if your fuses were applied using the CLI:

npx @electron/fuses list ./out/MyApp-win32-x64/MyApp.exe

Forge config.js do

|  |  |
| --- | --- |
| Section | Role |
| packagerConfig | Controls packaging behavior (e.g., ASAR bundling) |
| makers | Defines the output formats (e.g., .exe , .deb , .zip ) |
| plugins | Adds functionality like fuses or unpacking native modules |
| FusesPlugin | Locks down runtime behavior for better **security and integrity** |

## Electron forge

Electron Forge is a toolchain for:

|  |  |
| --- | --- |
| Task | Tool in Forge |
| Development | electron-forge start |
| Packaging | electron-forge package |
| Making installers | electron-forge make |
| Publishing | electron-forge publish |

It supports multiple targets (Windows, macOS, Linux) and simplifies release workflows.

## Node modules

In Electron, you can use **Node.js modules** just like in any Node app. These modules give you access to powerful backend features, such as:

**File system access** ( )



fs

**Path handling** ( )

path

**Network requests** ( ,

http

**Process and OS info** (

process

)

, )

https



os

sqlite3

Plus any package from **npm** (e.g.,

axios

,

, etc.)

## When to Use Electron Forge and Electron Forge Webpack

#### Use (Electron Forge) if:

**npx create-electron-app**

 You're a beginner and want to get started quickly.

 You want a simple project without messing with build tools like Webpack.  You're creating a quick prototype or small app.

 You want built-in support for building, packaging, and distributing your app with minimal setup.

 You donʼt need advanced features like hot reloading or custom Webpack plugins.

 If you are planning to go with HTML,CSS & JS alone then electron-forge is flexible

#### Use a Webpack (or Vite) template if:

 You need **Hot Module Replacement (HMR) or live reload** during development.

 You're building a **large or production-grade app** that needs performance optimization.

 You want full control over the build process (e.g., custom loaders, environment configs).

 Youʼre already familiar with Webpack/Vite or want to integrate advanced frontend frameworks (React, Vue, Svelte, etc.).

 Youʼre planning to split your app into multiple renderer processes or optimize assets.

## Dirty Form in Electron

### In Renderer Process (e.g., your form page)

let isFormDirty false;

// Set this to true when the user changes form data function onFormChange() {

isFormDirty true;

}

// Listen for the window close event

window.addEventListener('beforeunload', (e) ⇒ { if (isFormDirty) {

// Prevent the default close e.preventDefault();

// Chrome requires returnValue to be set e.returnValue '';

}

});

=

=

=

This will only show the default browser dialog ("Changes you made may not be saved."), which isn't customizable.

### Custom Dialog: Use IPC from Renderer ↔ Main

If you want a **custom confirmation dialog**, do this:

### Main Process (main.js)

const { app, BrowserWindow, dialog, ipcMain } = require('electron');

let win;

app.whenReady().then(() ⇒ { win = new BrowserWindow({

webPreferences: {

preload: dirname + '/preload.js', // preload for ipc

}

});

win.loadURL('file://' + dirname + '/index.html');

// Listen for confirmation request from renderer ipcMain.handle('confirm-exit', async () ⇒ {

const result = await dialog.showMessageBox(win, { type: 'question',

buttons: ['Exit without saving', 'Cancel'], defaultId: 1,

cancelId: 1,

message: 'Are you sure you want to exit without saving?',

});

return result.response  0; // true if 'Exit without saving'

});

win.on('close', async (e) ⇒ {

const { webContents } = win;

const shouldPreventClose = await webContents.executeJavaScript('windo w.isFormDirty === true');

if (shouldPreventClose) { e.preventDefault();

const confirm = await webContents.invoke('confirm-exit'); if (confirm) {

win.destroy(); // close forcefully

}

}

});

});

### Renderer Process (index.html or your JS)

<script>

window.isFormDirty false;

function onFormChange() { window.isFormDirty true;

}

// Simulate form input change

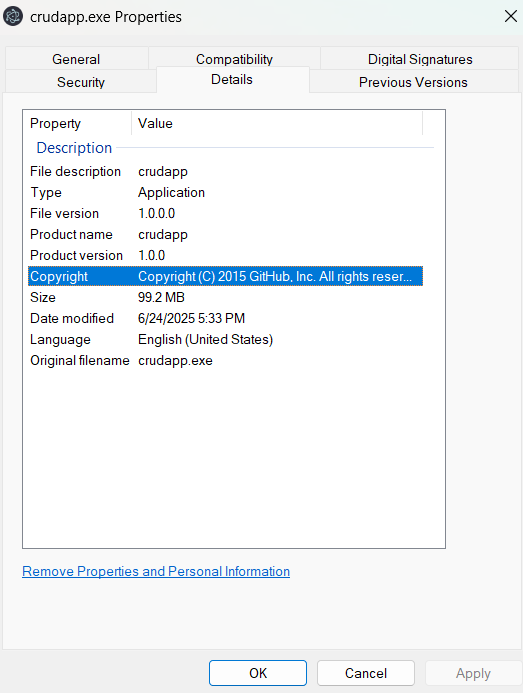
document.querySelector('input').addEventListener('input', onFormChange);

</script>

=

=

**Project Meta Data Configuration**



Setting up the project Meta Data

 In Non Web pack Forge electron project meta data will be taken directly from the package.json

 But Webpack projects it will use the Forge.Config.js meta data to be displayed on project detials

Example :

**package.json**

{

"name": "crudapp",

"version": "1.0.0",

"description": "A CRUD demo Electron app", "author": "Balaji Inc.",

"main": "main.js", "scripts": {

"start": "electron-forge start",

"make": "electron-forge make",

"package": "electron-forge package"

},

"config": {

"forge": "./forge.config.js"

},

"devDependencies": { "electron": "^28.0.0",

"@electron-forge/cli": "^6.0.0-beta.61"

}

}

Then run:

npx electron-forge import

This sets up basic Forge scaffolding for you (including makers,

.gitignore, etc.)

## Forge.Config.js

module.exports = { packagerConfig: {

icon: './assets/icon', // Without .ico win32metadata: {

CompanyName: 'Balaji Inc.', FileDescription: 'CRUD Desktop App', OriginalFilename: 'crudapp.exe', ProductName: 'crudapp',

InternalName: 'crudapp'

}

},

rebuildConfig: {}, makers: [

{

name: '@electron-forge/maker-squirrel', config: {

name: 'crudapp', authors: 'Balaji Inc.',

description: 'Electron CRUD desktop app', setupExe: 'crudapp-installer.exe', setupIcon: './assets/icon.ico'

}

}

]

};

Now run:

npm run make

Forge will:

 Package your app

Generate an installer and metadata based on

.exe

forge.config.js

Youʼll find your built files inside the directory

/out/make/

## Electron's Architecture: Main Process vs Renderer Process

|  |  |
| --- | --- |
| Layer | Role |
| **Main Process** | Controls app lifecycle, file system, native OS access |
| **Renderer** | Runs the UI (HTML/JS/CSS) — like a browser tab |
| **Preload** | Secure bridge between Main and Renderer via IPC |

My app has

#### → Main process

index.js

+ → **Renderer process**

index.html

renderer.js

\* Secure **IPC bridge**

preload.js

## IPC (Inter-Process Communication)

Because **renderer canʼt directly access Node APIs** (like

) for security, we use



fs

& :

ipcRenderer

ipcMain

|  |  |
| --- | --- |
| API | Use |
| ipcRenderer.send(channel, data) | From Renderer \* Main |
| ipcMain.on(channel, handler) | Listen in Main |

// In renderer.js

window.electronAPI.saveFormData(formData);

// In preload.js (safe bridge)

contextBridge.exposeInMainWorld('electronAPI', {

saveFormData: (data) ⇒ ipcRenderer.send('save-form-data', data)

});

// In index.js (main)

ipcMain.on('save-form-data', (event, formData) ⇒ {

// Save to data

});

## Workflow

### Renderer Process (

**renderer.js**

### or DOM)

This is where the **user interacts** (form, buttons, etc). You wrote:

window.electronAPI.saveFormData(formData);

Youʼre calling a function that does not exist in plain JS — it was exposed from the preload script.

### Preload Script ( )

This runs **before** the renderer loads. It's a "bridge" between the DOM and Node.js. You wrote:

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

contextBridge.exposeInMainWorld('electronAPI', {

saveFormData: (data) ⇒ ipcRenderer.send('save-form-data', data)

});

**preload.js**

#### What this does:

 Exposes a safe function

window.electronAPI.saveFormData(data)

 When called, it sends an IPC message (

save-form-data

It runs in a secure, isolated environment (thanks to

) to the **main process**

).

contextIsolation: true

### 📤 3. Sending Data via

**ipcRenderer.send()**

When the user clicks Save:

ipcRenderer.send('save-form-data', data);

This sends the form data to the **main process**, using a **channel** called .

'save-form-data'

### 🧠 4. Main Process ( )

The main process listens for that event:

**index.js**

ipcMain.on('save-form-data', (event, formData) ⇒ {

// formData is received here

// Now save it to a file using fs

});

## Full Workflow Summary

User Clicks Save Button]

↓ [renderer.js]

⤷ window.electronAPI.saveFormData(formData)

↓

[preload.js]

⤷ ipcRenderer.send('save-form-data', formData)

↓

[index.js (Main Process)]

⤷ ipcMain.on('save-form-data', callback)

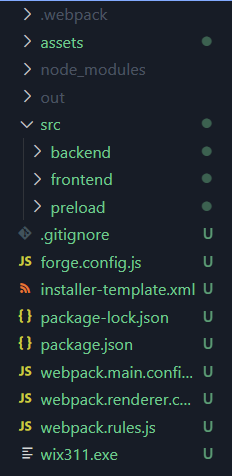
⤷ Save data to JSON file

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Used In | Direction | Description |
| ipcRenderer.send(channel, ...args) | Renderer Process | Renderer ➜  Main | Sends an asynchronous message to the main process (fire-and-forget). |
| ipcMain.on(channel, listener) | Main Process | Renderer ➜  Main | Listens for asynchronous messages sent via  ipcRenderer.send . |

|  |  |  |  |
| --- | --- | --- | --- |
| ipcRenderer.invoke(channel, ...args) | Renderer Process | Renderer ➜  Main (awaited) | Sends a request to the main process and  expects a response (uses  await ). |
| ipcMain.handle(channel, handler) | Main Process | Renderer ➜  Main (awaited) | Handles an IPC request initiated by  ipcRenderer.invoke . |
| ipcMain.emit(channel, ...args) | Main Process | Internal | Emits a custom event — not commonly used for renderer-main comms. |
| ipcRenderer.on(channel, listener) | Renderer Process | Main ➜  Renderer | Listens for messages from the main process sent using  webContents.send . |
| webContents.send(channel, ...args) | Main Process | Main ➜  Renderer | Sends an async message to a renderer process. |
| ipcMain.handleOnce(channel, handler) | Main Process | Renderer ➜  Main (awaited) | Like handle , but runs only once. |
| ipcRenderer.once(channel, listener) | Renderer Process | Main ➜  Renderer | Listens once for a message from the main process. |
| ipcRenderer.removeListener(channel, listener) | Renderer Process | ———- | Removes a specific listener for the given  channel. |
| ipcRenderer.removeAllListeners(channel) | Renderer Process |  | Removes all  listeners for the  specified channel. |

**Folder Structure Enhancement**

To keep the folder structure clean, maintainable, and scalable, it's important to separate concerns and organize files logically.



sample folder structure

Configure the main.js file path as a entry point on the webpack.main.config.js

module.exports {

entry: './src/backend/main.js',

// Put your normal webpack config below here target: 'electron-main',

module: {

rules: require('./webpack.rules'),

},

resolve: {

extensions: ['.js', '.json'],

=

},

externals: {

'better-sqlite3': 'commonjs better-sqlite3'

}

};

Configure the html index page and js renderer page and the preload.js file path as a entry point on the forge.config.js

name: '@electron-forge/plugin-webpack', config: {

mainConfig: './webpack.main.config.js', renderer: {

config: './webpack.renderer.config.js', entryPoints: [

{

html: './src/frontend/index.html', js: './src/frontend/renderer.js', name: 'main\_window',

preload: {

js: './src/preload/preload.js',

},

},

],

## Sqlite3 Configuration Install Dependencies

Make sure you have installed:

sqlite3

npm install sqlite3

If you're using Electron and run into build issues (native module), you might need:\

npm install sqlite3 --build-from-source

Or use electron-rebuild:

npm install --save-dev electron-rebuild npx electron-rebuild

## Import Electron package

const sqlite3  require('sqlite3').verbose();

import sqlite package in the js file

The .verbose() method **enables long stack traces and extra debugging**

**information**. Itʼs primarily useful for **development and debugging**, not production.

const path = require('path');

const sqlite3  require('sqlite3').verbose();

const dbPath = path.join( dirname, 'mydatabase.db'); const db = new sqlite3.Database(dbPath)

Start the db usage with db.serialize() or db.parallelize()

|  |  |
| --- | --- |
| Method | Purpose |
| db.serialize() | Ensures that queries inside the callback run **sequentially**. |
| db.parallelize() | Ensures that queries inside the callback run **in parallel**  (default). |

## Db methods

|  |  |
| --- | --- |
| Method | Purpose |
| db.run() | Execute SQL that **doesn't return rows**, like INSERT , UPDATE , DELETE , or  CREATE TABLE . |
| db.get() | Execute a SELECT query and return **the first row** only. |
| db.all() | Execute a SELECT query and return **all matching rows** as an array. |

|  |  |
| --- | --- |
| db.each() | Execute a SELECT query and run a callback **for each row**, one at a time. |
| db.exec() | Run **multiple SQL statements** at once (separated by ; ), no results returned. |

Example :

db.get('SELECT \* FROM users WHERE email  ?', [email]);

To run multiple times

db.run('INSERT INTO users (name, email, password) VALUES (?, ?, ?)', [name, e mail, hashedPassword]);

## Prepared Statements

When you call , it returns a **Statement** object with its own methods:

db.prepare()

|  |  |
| --- | --- |
| Statement Method | Purpose |
| stmt.run() | Execute the prepared SQL with bound parameters. |
| stmt.get() | Get a single row (like db.get ). |
| stmt.all() | Get all rows (like db.all ). |
| stmt.finalize() | Free up resources after the prepared statement is no longer needed. |
| stmt.bind() | Bind parameters to the statement without running it yet. |

Example :

statement db.prepare('INSERT INTO users (name, email) VALUES (?, ?)'); stmt.run(['Balaji', ['balaji@gmail.com'](mailto:%27balaji@gmail.com)]);

stmt.run(['Jawath', ['jawathali@gmail.com'](mailto:%27jawathali@gmail.com)]);

=

so that you can execute multiple values with the single query

const selectStmt db.prepare('SELECT \* FROM users WHERE email  ?'); selectStmt.get([['balaji@gmail.com'](mailto:%27balaji@gmail.com)], (err, row) ⇒ {

if (err) return console.error(err.message);

=

console.log('Selected:', row); selectStmt.finalize();

### Finalize()

All the statements should be finalized using .finalize(); method

// Example

selectStmt.finalize();

Call the function on the separate backend file

const insertUser

try {

async (event,name, email, password)

{

const result db.insertUser(name, email, password); return result;

} catch (error) {

console.error('Error inserting user:', error.message); throw error;

}

}

=

⇒

=

Import the functions and handle it on the main file using ipcMain()

ipcMain.handle('db:getUsers', api.getUsers); ipcMain.handle('db:insertUser', api.insertUser); ipcMain.handle('db:loginUser', api.loginUser); ipcMain.handle('db:updateUser', api.updateUser); ipcMain.handle('db:deleteUser', api.deleteUser);

Using Preload Invoke the Ipc Calls as ipcRenderer()

contextBridge.exposeInMainWorld('api', {

insertUser: (name, email, password ) ⇒ ipcRenderer.invoke('db:insertUser', na me, email, password),

getUsers: () ⇒ ipcRenderer.invoke('db:getUsers'),

updateUser: (id, name, email) ⇒ ipcRenderer.invoke('db:updateUser', id, nam e, email),

loginUser: (email, password) ⇒ ipcRenderer.invoke('db:loginUser', email, pass word),

deleteUser: (id) ⇒ ipcRenderer.invoke('db:deleteUser', id),

});

Call the function on the front end js files using window

await window.api.insertUser(name, email, password);

## Adding the Installation File with an Interface (NSIS)

### 🔨 1. Build Your App Executable

Use the following command to generate the executable of your Electron app (or desktop app):

npm run make

This will output your app's .exe and supporting files (usually inside the out/ or dist/ directory).

### 2. Install NSIS (Installer Generator)

Download and install **NSIS (Nullsoft Scriptable Install System)** from the official site: <https://nsis.sourceforge.io/Download>

Make sure the command is added to your system PATH during installation.

makensis

### 📝 3. Create the Installer Script

**.nsi**

YourInstaller.nsi

TrucsInstaller.nsi

Create a file named Example:

!include "MUI2.nsh" Name "Your App"

OutFile "YourInstaller.exe"

(e.g.,

) and define the installer logic.

InstallDir "$PROGRAMFILES\YourApp"

RequestExecutionLevel admin

!insertmacro MUI\_PAGE\_WELCOME

!insertmacro MUI\_PAGE\_DIRECTORY

!insertmacro MUI\_PAGE\_INSTFILES

!insertmacro MUI\_PAGE\_FINISH

!insertmacro MUI\_LANGUAGE "English"

Section "Install" SetOutPath "$INSTDIR"

File /r "..\out\YourApp-win32-x64\\*"

WriteUninstaller "$INSTDIR\Uninstall.exe" SectionEnd

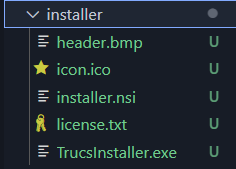
### 4. Run NSIS Build Command

Use the following command to compile the installer using NSIS:

makensis YourInstaller.nsi

This will generate YourInstaller.exe, a complete installer with a user-friendly interface.

## Installer files



Folder structure

### .nsi file

A (Nullsoft Scriptable Installer) file is a **script used by NSIS (Nullsoft Scriptable**



.nsi

**Install System)** to define how a Windows installer should behave. It contains instructions for:

 What files to install and where

 What UI pages to show (welcome, license, directory, etc.)  How to create shortcuts, registry entries, or uninstallers

 Any custom logic during install/uninstall

Think of it as the blueprint for your installer — it controls the interface and functionality of the setup .exe.

Sample .nsi code

!include "MUI2.nsh"

!include "LogicLib.nsh"

;

BASIC INFO

Name "DB Setup App - Trucs Company" OutFile "TrucsInstaller.exe"

InstallDir "$PROGRAMFILES\DBSetupApp"

InstallDirRegKey HKCU "Software\DBSetupApp" "Install\_Dir" RequestExecutionLevel admin



==

; VERSION INFO



==

VIProductVersion "1.0.0.0"

VIAddVersionKey "ProductName" "DB Setup App"

VIAddVersionKey "CompanyName" "Trucs Company" VIAddVersionKey "FileVersion" "1.0.0.0"

VIAddVersionKey "FileDescription" "Database Setup Application Installer"

; UI CONFIGURATION



==

!define MUI\_ABORTWARNING

; ==Installation steps======

!insertmacro MUI\_PAGE\_WELCOME

!insertmacro MUI\_PAGE\_LICENSE "license.txt"

!insertmacro MUI\_PAGE\_DIRECTORY

!insertmacro MUI\_PAGE\_COMPONENTS

!insertmacro MUI\_PAGE\_INSTFILES

!insertmacro MUI\_PAGE\_FINISH

; ========Uninstall steps

==========

!insertmacro MUI\_UNPAGE\_WELCOME

!insertmacro MUI\_UNPAGE\_CONFIRM

!insertmacro MUI\_UNPAGE\_INSTFILES

!insertmacro MUI\_UNPAGE\_FINISH

!insertmacro MUI\_LANGUAGE "English"

; COMPONENTS



==

Section "!DB Setup App (Required)" SecMain SectionIn RO

SetOutPath "$INSTDIR"

; Copy packaged application files

File /r "..\out\DB Setup App-win32-x64\\*"

; Write uninstaller

WriteUninstaller "$INSTDIR\Uninstall.exe"

; Registry entries

WriteRegStr HKLM "Software\Microsoft\Windows\CurrentVersion\Uninstall\DB SetupApp" "DisplayName" "DB Setup App"

WriteRegStr HKLM "Software\Microsoft\Windows\CurrentVersion\Uninstall\DB SetupApp" "UninstallString" "$INSTDIR\Uninstall.exe"

WriteRegStr HKLM "Software\Microsoft\Windows\CurrentVersion\Uninstall\DB SetupApp" "Publisher" "Trucs Company"

WriteRegStr HKLM "Software\Microsoft\Windows\CurrentVersion\Uninstall\DB SetupApp" "DisplayVersion" "1.0.0"

SectionEnd

Section "Desktop Shortcut" SecDesktop

; Look for the main executable with different possible names IfFileExists "$INSTDIR\dbsetupapp.exe" 0 TryOtherName

CreateShortcut "$DESKTOP\DB Setup App.lnk" "$INSTDIR\dbsetupapp.exe" Goto ShortcutCreated

TryOtherName:

IfFileExists "$INSTDIR\DB Setup App.exe" 0 ShortcutCreated

CreateShortcut "$DESKTOP\DB Setup App.lnk" "$INSTDIR\DB Setup App.ex

e"

ShortcutCreated:

SectionEnd

Section "Start Menu Shortcuts" SecStartMenu CreateDirectory "$SMPROGRAMS\DB Setup App"

; Look for the main executable with different possible names IfFileExists "$INSTDIR\dbsetupapp.exe" 0 TryOtherNameSM

CreateShortcut "$SMPROGRAMS\DB Setup App\DB Setup App.lnk" "$INSTD IR\dbsetupapp.exe"

Goto SMShortcutCreated TryOtherNameSM:

IfFileExists "$INSTDIR\DB Setup App.exe" 0 SMShortcutCreated

CreateShortcut "$SMPROGRAMS\DB Setup App\DB Setup App.lnk" "$INSTD IR\DB Setup App.exe"

SMShortcutCreated:

CreateShortcut "$SMPROGRAMS\DB Setup App\Uninstall.lnk" "$INSTDIR\Uni nstall.exe"

SectionEnd

; DESCRIPTIONS



==

!insertmacro MUI\_FUNCTION\_DESCRIPTION\_BEGIN

!insertmacro MUI\_DESCRIPTION\_TEXT $SecMain} "Main application files (re quired)"

!insertmacro MUI\_DESCRIPTION\_TEXT $SecDesktop} "Create a shortcut on t he desktop"

!insertmacro MUI\_DESCRIPTION\_TEXT $SecStartMenu} "Create shortcuts in the Start Menu"

!insertmacro MUI\_FUNCTION\_DESCRIPTION\_END

; UNINSTALLER



==

Section "Uninstall"

Delete "$INSTDIR\\*.\*" RMDir /r "$INSTDIR"

Delete "$DESKTOP\DB Setup App.lnk" RMDir /r "$SMPROGRAMS\DB Setup App"

DeleteRegKey HKLM "Software\Microsoft\Windows\CurrentVersion\Uninstall

\DBSetupApp"

DeleteRegKey HKCU "Software\DBSetupApp" SectionEnd

Function .onInstSuccess

MessageBox MB\_ICONINFORMATION "DB Setup App has been successfully i nstalled!"

FunctionEnd

## Includes and Setup

**MUI2.nsh**: Imports **Modern UI 2** macros for building user-friendly interface pages (welcome, license, etc.).

IfFileExists

StrCmp

ifelse

 **LogicLib.nsh**: Adds advanced flow control (like

## Basic Information

,

,

etc.).

: Title shown in the installer window and task manager.

Name "DB Setup App - Trucs Company" OutFile "TrucsInstaller.exe"

InstallDir "$PROGRAMFILES\DBSetupApp"

InstallDirRegKey HKCU "Software\DBSetupApp" "Install\_Dir" RequestExecutionLevel admin

Name

: Name of the installer output.

OutFile

.exe

: Default installation folder (e.g., ).

InstallDir

C:\Program Files\DBSetupApp

: Remembers the install path for future runs by storing it in registry.

VIProductVersion "1.0.0.0"

VIAddVersionKey "ProductName" "DB Setup App"

VIAddVersionKey "CompanyName" "Trucs Company" VIAddVersionKey "FileVersion" "1.0.0.0"

VIAddVersionKey "FileDescription" "Database Setup Application Installer"

InstallDirRegKey

: Ensures the installer runs with **admin privileges** (needed for Program Files, registry, etc.).

RequestExecutionLevel admin

## Version Info for File Properties

These lines define **metadata embedded in the**

include "MUI2.nsh"

!include "LogicLib.nsh"

**.exe**

Properties \* Details).

**file** (shown in right-click \*

## UI Configuration

!define MUI\_ABORTWARNING

 Displays a **"Do you really want to quit?"** warning if the user clicks cancel during installation.

## Installation Pages (Displayed to User)

!insertmacro MUI\_PAGE\_WELCOME

!insertmacro MUI\_PAGE\_LICENSE "license.txt"

!insertmacro MUI\_PAGE\_DIRECTORY

!insertmacro MUI\_PAGE\_COMPONENTS

!insertmacro MUI\_PAGE\_INSTFILES

!insertmacro MUI\_PAGE\_FINISH

Each macro adds a prebuilt page to the installer wizard:

 **Welcome**: Greets the user

 **License**: Shows license agreement from

license.txt

 **Directory**: Lets user choose install folder

 **Components**: Lets user select optional features (sections)

 **InstFiles**: Shows file copy progress

 **Finish**: Displays success/final options (e.g., "Run app")

## Uninstaller Pages

!insertmacro MUI\_UNPAGE\_WELCOME

!insertmacro MUI\_UNPAGE\_CONFIRM

!insertmacro MUI\_UNPAGE\_INSTFILES

!insertmacro MUI\_UNPAGE\_FINISH

Similar to install pages, but for **uninstallation** flow.

## Language

!insertmacro MUI\_LANGUAGE "English"

 Sets the installer language to English (UI strings, messages, buttons, etc.).

## SECTION: Main App Installation

Creates a **required section** named "DB Setup App".

Section "!DB Setup App (Required)" SecMain SectionIn RO

: Makes it **read-only** (cannot be unchecked in component list).

SectionIn RO

Sets the target folder to the chosen install path ( ).

SetOutPath "$INSTDIR"

$INSTDIR

Recursively copies the built app files from the specified folder into the install directory.

File /r "..\out\DB Setup App-win32-x64\\*"

Generates an uninstaller in the install folder.

WriteUninstaller "$INSTDIR\Uninstall.exe"

; Registry entries

WriteRegStr HKLM "Software\Microsoft\Windows\CurrentVersion\Uninstall\DB SetupApp" "DisplayName" "DB Setup App"

WriteRegStr HKLM "Software\Microsoft\Windows\CurrentVersion\Uninstall\DB SetupApp" "UninstallString" "$INSTDIR\Uninstall.exe"

WriteRegStr HKLM "Software\Microsoft\Windows\CurrentVersion\Uninstall\DB SetupApp" "Publisher" "Trucs Company"

WriteRegStr HKLM "Software\Microsoft\Windows\CurrentVersion\Uninstall\DB SetupApp" "DisplayVersion" "1.0.0"

 Adds **Add/Remove Programs** entry so Windows can manage uninstallation.

## SECTION: Desktop Shortcut

Optional section that adds a shortcut to the desktop.

Section "Desktop Shortcut" SecDesktop

IfFileExists "$INSTDIR\dbsetupapp.exe" 0 TryOtherName

CreateShortcut "$DESKTOP\DB Setup App.lnk" "$INSTDIR\dbsetupapp.exe" Goto ShortcutCreated

TryOtherName:

IfFileExists "$INSTDIR\DB Setup App.exe" 0 ShortcutCreated

CreateShortcut "$DESKTOP\DB Setup App.lnk" "$INSTDIR\DB Setup App.ex

e"

ShortcutCreated:

Checks for either

dbsetupapp.exe

or

in the install directory and

creates a shortcut accordingly.

DB Setup App.exe

## SECTION: Start Menu Shortcuts

Adds app and uninstaller shortcuts to the **Start Menu**.

Section "Start Menu Shortcuts" SecStartMenu

Creates a Start Menu folder.

CreateDirectory "$SMPROGRAMS\DB Setup App"

IfFileExists "$INSTDIR\dbsetupapp.exe" 0 TryOtherNameSM

CreateShortcut "$SMPROGRAMS\DB Setup App\DB Setup App.lnk" "$INSTD

IR\dbsetupapp.exe"

Goto SMShortcutCreated TryOtherNameSM:

IfFileExists "$INSTDIR\DB Setup App.exe" 0 SMShortcutCreated

CreateShortcut "$SMPROGRAMS\DB Setup App\DB Setup App.lnk" "$INSTD IR\DB Setup App.exe"

SMShortcutCreated:

CreateShortcut "$SMPROGRAMS\DB Setup App\Uninstall.lnk" "$INSTDIR\Uni nstall.exe"

 Same logic as desktop shortcut, but places links in Start Menu.

## Component Descriptions

!insertmacro MUI\_FUNCTION\_DESCRIPTION\_BEGIN

!insertmacro MUI\_DESCRIPTION\_TEXT $SecMain} "Main application files (re quired)"

!insertmacro MUI\_DESCRIPTION\_TEXT $SecDesktop} "Create a shortcut on t he desktop"

!insertmacro MUI\_DESCRIPTION\_TEXT $SecStartMenu} "Create shortcuts in the Start Menu"

!insertmacro MUI\_FUNCTION\_DESCRIPTION\_END

 Descriptions shown when hovering over components in the component selection page.

## SECTION: Uninstaller Logic

Defines the actions the uninstaller will perform.

Section "Uninstall"

Delete "$INSTDIR\\*.\*" RMDir /r "$INSTDIR"

Delete "$DESKTOP\DB Setup App.lnk"

RMDir /r "$SMPROGRAMS\DB Setup App"

DeleteRegKey HKLM "Software\Microsoft\Windows\CurrentVersion\Uninstall

\DBSetupApp"

DeleteRegKey HKCU "Software\DBSetupApp"

 Deletes installed files, desktop & Start Menu shortcuts, and cleans up registry entries.

## Post-Install Message

Function .onInstSuccess

MessageBox MB\_ICONINFORMATION "DB Setup App has been successfully i nstalled!"

FunctionEnd

 Displays a confirmation popup after successful installation.

## Packages and thier usages

### Electron Package

**Components used:** BrowserWindow, Menu, app, ipcMain **Purpose:** Core Electron framework for building desktop apps **Where used:** Throughout the entire application

const {

BrowserWindow, Menu,

app, ipcMain

} require('electron');

=

**Why:** Essential for creating the desktop application window, menus, and inter- process communication

**Effect without it:** Application cannot exist - it's the foundation

### electron-context-menu

const contextMenu require('electron-context-menu');

=

 **Purpose:** Provides right-click context menu functionality

#### Where used:

contextMenu({

showLookUpSelection: false, showSearchWithGoogle: false, showCopyImage: false,

showCopyImageAddress: false, showSaveImage: false,

showSaveImageAs: false, showInspectElement: true, showServices: false

}); // can be added for right click functionality

//or

/\*

mainWindow.webContents.on('context-menu', (event, params) ⇒ { const menu  Menu.buildFromTemplate([

{ label: 'Cut', role: 'cut' },

{ label: 'Copy', role: 'copy' },

{ label: 'Paste', role: 'paste' },

{ label: 'Delete', role: 'delete' },

{ type: 'separator' },

{ label: 'Select All', role: 'selectall' },

{ type: 'separator' },

{ label: 'Inspect Element', click: () ⇒ {

mainWindow.webContents.inspectElement(params.x, params.y);

}

}

]);

menu.popup();

}); //working code

\*/

 **Why:** Enables copy/paste/inspect functionality in input fields

 **Effect without it:** No right-click menus, users can't copy/paste text

### electron-context-menu

**Purpose:** Enables Dev Tools in production builds

const debug require('electron-debug');

=

#### Where used:

debug({

isEnabled: true, showDevTools

});

 **Why:** Allows troubleshooting in production without auto-showing DevTools

 **Effect without it:** No Dev Tools access in production builds for debugging

### 4. electron-is-dev

**Purpose:** Detects if app is running in development mode

const isDev require('electron-is-dev');

=

#### Where used :

if (isDev) {

require('electron-reload')(path.join( dirname, 'build'));

} // When in development mode: - Enable automatic reloads

// Path to root directory.

const basePath isDev ? dirname : app.getAppPath();

=

if (isDev) {

mainWindow.webContents.session.clearCache();

}// clears the session cache

 **Why:** Enables development-specific features (hot reload, cache clearing, WebRTC internals)

 **Effect without it:** Cannot distinguish dev/prod environments, no dev-specific features

### electron-updater

**Purpose:** Handles automatic app updates

const { autoUpdater } require('electron-updater');

=

#### Where used:

autoUpdater.logger = require('electron-log'); // used to log the updated log info or failed information

autoUpdater.logger.transports.file.level = 'info';

/\* Was an update available? Did it download successfully? Was there an error?

Was the update applied \*/

// .Mac App Store dont allow auto update so if its mac then notify and allow u pdate from the app store

if (!process.mas) {

autoUpdater.checkForUpdatesAndNotify();

}

 **Why:** Keeps the app updated automatically for better security and features

 **Effect without it:** No automatic updates, users must manually download new versions

### electron-window-state

**Purpose:** Remembers and restores window size/position

const windowStateKeeper require('electron-window-state');

=

#### Where used:

const windowStateKeeper = require('electron-window-state');

let mainWindowState = windowStateKeeper({ defaultWidth: 800,

defaultHeight: 600

});

const mainWindow = new BrowserWindow({ x: mainWindowState.x,

y: mainWindowState.y,

width: mainWindowState.width, height: mainWindowState.height,

});

mainWindowState.manage(mainWindow);

 **Why:** Better UX by restoring user's preferred window layout

 **Effect without it:** Window always opens at default size/position

## Node.js Core Modules

### path

const path require('path');

=

**Purpose:** File path manipulation utilities

#### Where used:

const newpath path.join( dirname ,"newfolder",file.txt);

=

preload: path.resolve(basePath, './build/preload.js')

const requestedPath = path.resolve(URL.fileURLToPath(details.url)); const appBasePath = path.resolve(basePath);

 **Why:** Safely constructs file paths across different operating systems

 **Effect without it:** Path resolution errors, potential security vulnerabilities

### process

**Purpose:** Access to current Node.js process information

const process require('process');

=

#### Where used :

**Why:** Platform detection, command line arguments, environment variables

if (isDev && process.platform 'win32') { app.setAsDefaultProtocolClient(

config.default.appProtocolPrefix, process.execPath,

[ path.resolve(process.argv[1]) ]

);

===

 **Effect without it:** Cannot detect OS, handle command line args, or manage app lifecycle

### URL

**Purpose:** URL parsing and formatting utilities

const URL require('url');



#### Where used:

const requestedPath path.resolve(URL.fileURLToPath(details.url));

=

// URL for index.html which will be our entry point. const indexURL URL.format({

pathname: path.resolve(basePath, './build/index.html'), protocol: 'file:',

slashes: true

});



 **Why:** Safely constructs file URLs and parses external URLs

 **Effect without it:** Incorrect URL handling, potential security issues.

## Db Backup

Set the backup path and backup file on the user data directory

const dbPath path.join(app.getPath('userData'), 'database.db');

const backupPath path.join(app.getPath('userData'), 'database\_backup.db');

=

=

Write a function for copying the backup file

function createBackup() {

if (fs.existsSync(dbPath)) {

try {

fs.copyFileSync(dbPath, backupPath);

console.log('Database backup updated at:', backupPath);

} catch (error) {

console.error('Failed to update backup:', error.message);

}

}

}

Call the Create backup function on all the updation so that for every update it will update the backup file

function updateUser(id, name, email) { return new Promise((resolve, reject) ⇒ {

db.run('UPDATE users SET name  ?, email  ? WHERE id  ?', [name, email, id], function(err) {

if (err) {

console.error('Error updating user:', err.message); reject(err);

} else {

console.log(`User with ID ${id} updated successfully.`);

// Create backup after successful update createBackup();

resolve({ success: true });

}

});

# Storage

## Local Storage

Data will be there even reloaded the page or seen after few days or even if closes the session and apps

To Store

//localStorage.setItem('key', 'value');

window.localStorage.setItem('FName','FName');

To retrive Session Storage

Data will be stored on the session until the session gets over , if the tab/applications closed and reloaded then the session data will be

//localStorage.getItem('key');

let fname window.localStorage.getItem("FName");

=

To Store

//sessionStorage.setItem('key', 'value');

window.sessionStorage.setItem('FName','FName');

To retrive

//sessionStorage.getItem('key');

let fname window.sessionStorage.getItem("FName");

=

[**Create an MSI file for y our Electron app on Windows using Electron**](https://www.electronforge.io/config/makers/wix-msi)[**Forge**](https://www.electronforge.io/config/makers/wix-msi)**.**

The WiX MSI target builds files, which are "traditional" Windows installer files.

.msi

We generally recommend using the [Squirrel.Windows](https://www.electronforge.io/config/makers/squirrel.windows) target over using this one. These MSI files are a worse user experience for installation but sometimes it is necessary to build MSI files to appease large-scale enterprise companies with

internal application distribution policies.

## Requirements

You can only build the WiX MSI target on machines with [WiX Toolset v3](https://wixtoolset.org/docs/wix3/) installed. We recommend pinning your installation of WiX Toolset to a specific version. You can install WiX Toolset on Windows via [Chocolatey](https://chocolatey.org/).

Copy

choco install wixtoolset --version=3.14.0

## Installation

Copy

npm install --save-dev @electron-forge/maker-wix

## Usage

To use Copy

module.exports = { makers: [

{

name: '@electron-forge/maker-wix', config: {

language: 1033,

manufacturer: 'My Awesome Company'

}

}

]

};

@electron-forge/maker-wix

, add it to the

array in your [Forge configuration](https://www.electronforge.io/config/configuration):

makers

# Dark Mode Overview

### Automatically update the native interfaces

const { app, BrowserWindow, ipcMain, nativeTheme } require('electron/mai n')

=

"Native interfaces" include the file picker, window border, dialogs, context menus, and more - anything where the UI comes from your operating system and not from your app. The default behavior is to opt into this automatic theming from the OS.

//Main.js

ipcMain.handle('dark-mode:toggle', () ⇒ { if (nativeTheme.shouldUseDarkColors) {

nativeTheme.themeSource

} else {

nativeTheme.themeSource

}

'light'

'dark'

return nativeTheme.shouldUseDarkColors

})

ipcMain.handle('dark-mode:system', () ⇒ { nativeTheme.themeSource 'system'

})

=

=

=

//Preload.js

contextBridge.exposeInMainWorld('darkMode', { toggle: () ⇒ ipcRenderer.invoke('dark-mode:toggle'), system: () ⇒ ipcRenderer.invoke('dark-mode:system')

})

!-- html ⟶

<button id="toggle-dark-mode"Toggle Dark Mode</button>

<button id="reset-to-system"Reset to System Theme</button>

//renderer.js

document.getElementById('toggle-dark-mode').addEventListener('click', async

() ⇒ {

const isDarkMode await window.darkMode.toggle()

document.getElementById('theme-source').innerHTML isDarkMode ? 'Dark'

: 'Light'

})

document.getElementById('reset-to-system').addEventListener('click', async ()

⇒ {

await window.darkMode.system()

document.getElementById('theme-source').innerHTML 'System'

})





=

/\* Index.css \*/

:root {

color-scheme: light dark;

}

@media (prefers-color-scheme: dark) { body { background: #333; color: white; }

}

@media (prefers-color-scheme: light) { body { background: #ddd; color: black; }

}

## Electron Cloud Sync

Set the tray icon for the sync up status

**src/main/main.js**

const { app, Tray, Menu } require('electron')

const path require('path')

let tray null

app.whenReady().then(() ⇒ {

=

=

=

tray new Tray(path.join( dirname, '../../public/tray-icon.png'))

const contextMenu Menu.buildFromTemplate([

{ label: 'Sync Now', click: () ⇒ console.log('Manual sync...') },

{ label: 'Quit', click: () ⇒ app.quit() }

])

tray.setContextMenu(contextMenu)

tray.setToolTip('SmartSync running in background')

// 🔁 Add watcher and sync logic later

})



=

Use the chokidar for the changes made in the folders , the below program will give me all the changes on downloads/sample folder.

const chokidar require('chokidar');

const path require('path');

const watchFolder path.join(process.env.HOME, 'Downloads','sample');

chokidar.watch(watchFolder, { persistent: true, ignoreInitial: true, //

ignored: /(^|\/\\])\../

})

.on('add', (path) ⇒ console.log(`File ${path} has been added`))

.on('change', (path) ⇒ console.log(`File ${path} has been changed`))

.on('unlink', (path) ⇒ console.log(`File ${path} has been removed`));

=

=

=

### Connect to

**main.js**

**Upload on Azure**

Open your

src/main/main.js

and add this line:

require('./watcher') // ✅ add at the bottom

So it looks like:

tray.setToolTip('SmartSync running')

// 🧩 Add file watcher require('./watcher')

📄

const  BlobServiceClient } = require('@azure/storage-blob') const path = require('path')

src/main/azureUpload.js

const fs = require('fs')

const AZURE\_CONNECTION\_STRING  'Your\_Azure\_Connection\_String\_Her e'

const CONTAINER\_NAME  'smartsync-container'

async function uploadFileToAzure(filePath) { try {

const blobServiceClient  BlobServiceClient.fromConnectionString(AZUR E\_CONNECTION\_STRING)

const containerClient = blobServiceClient.getContainerClient(CONTAINER

\_NAME)

const blobName = path.basename(filePath)

const blockBlobClient = containerClient.getBlockBlobClient(blobName)

await blockBlobClient.uploadFile(filePath)

console.log(`✅ Uploaded to Azure: ${blobName}`)

} catch (err) {

console.error('❌ Azure upload failed:', err.message) throw err

}

}

module.exports = { uploadFileToAzure }

## ✅ Step 9.4: Connect to the Watcher

Now go to:

📄

src/main/watcher.js

At the top:

.on('change')

const { uploadFileToAzure } = require('./azureUpload')

In

.on('add')

and

, update like this:

.env

dotenv

azureUpload.js

.env

.on('add', (file) ⇒ {

console.log('📁 File added:', file) uploadFileToAzure(file)

})

.on('change', (file) ⇒ {

console.log('✏ File changed:', file) uploadFileToAzure(file)

})

Move your Azure connection string into a file and use :

npm install dotenv

Then in :

require('dotenv').config()

const AZURE\_CONNECTION\_STRING  process.env.AZURE\_CONN

And in file (in root):

AZURE\_CONN=your-connection-string-here

## Create

**syncLogic.js**

📄

src/main/syncLogic.js

const fs = require('fs')

const dns = require('dns')

const path require('path')

=

const { uploadFileToAzure } require('./azureUpload')

=

const cachePath path.join( dirname, 'sync-cache.json')

=

// 🔁 Check if online (simple DNS check) function isOnline(callback) {

dns.lookup('google.com', (err) ⇒ { callback(!err)

})

}

// 🔐 Store file path in cache file function cacheFile(filePath) {

let cache []

=

if (fs.existsSync(cachePath)) { try {



cache JSON.parse(fs.readFileSync(cachePath))

} catch (e) {

cache []

=

}

}

if (!cache.includes(filePath)) { cache.push(filePath)

fs.writeFileSync(cachePath, JSON.stringify(cache, null, 2))

console.log('🕓 File cached for later sync:', filePath)

}

}

// 🔁 Called from watcher on every change

function handleFileChange(filePath) { isOnline((online) ⇒ {

if (online) {

uploadFileToAzure(filePath).catch(() ⇒ { cacheFile(filePath)

})

} else { cacheFile(filePath)

}

})

}

module.exports { handleFileChange, retryCachedSync }

=

## Add Retry Sync Function

Still in , add this function at the bottom:

syncLogic.js

function retryCachedSync() { isOnline(async (online) ⇒ {

if (!online) return

if (!fs.existsSync(cachePath)) return

const cache  JSON.parse(fs.readFileSync(cachePath)) if (cache.length  0) return

console.log('🌐 Net back! Syncing cached files...')

const remaining = []

for (const file of cache) { try {

await uploadFileToAzure(file)

console.log('✅ Synced cached file:', file)

} catch (err) {

console.log('⚠ Still failed:', file) remaining.push(file)

}

}

fs.writeFileSync(cachePath, JSON.stringify(remaining, null, 2))

})

}

module.exports = { handleFileChange, retryCachedSync }

## Update

📄 In

src/main/watcher.js

## to Use This

, update:

**watcher.js**

const { handleFileChange, retryCachedSync } = require('./syncLogic')

// Inside watcher watcher

.on('add', (file) ⇒ {

console.log('📁 File added:', file) handleFileChange(file)

})

.on('change', (file) ⇒ {

console.log('✏ File changed:', file) handleFileChange(file)

})

## ✅ Step 10.4: Schedule Retry from

📄 In , add:

**main.js**

src/main/main.js

const { retryCachedSync } = require('./syncLogic') setInterval(retryCachedSync, 30000)

You can place that line just after tray is set up.