Electron Course

Flavio Copes

Updated August 6, 2026

Contents

Learn to build secure desktop applications with Electron, Forge, BrowserWindow, preload scripts, IPC, local files, native menus, packaging, and signing.

What you'll learn: Build, secure, package, and prepare a local-first desktop notes application for distribution on macOS, Windows, and Linux.

Foundations and setup

Choose Electron deliberately and create a project with Electron Forge.

What Electron is

Understand how Electron combines Chromium and Node.js to run one JavaScript application on macOS, Windows, and Linux.

Electron combines Chromium, Node.js, and desktop operating-system APIs in one application runtime.

Chromium renders HTML and CSS. Node.js and Electron APIs let privileged code work with files, windows, menus, and native dialogs. Your users install the application, not Node.js or a separate browser.

This does not turn every page into one powerful JavaScript environment. Electron uses several processes and security boundaries. The page stays in a renderer. Privileged work belongs in the main process. A preload script exposes the small bridge between them.

Shipping the runtime gives us consistent web capabilities across macOS, Windows, and Linux. It also has costs:

- the application contains a browser engine, so downloads and memory use are larger
- every supported platform needs packaging and testing
- the application must keep Electron current for Chromium and Node.js security fixes
- a renderer security bug can become a desktop security bug if privileges are exposed carelessly

During this course we will build **Desktop Notes**. It will store local notes, use a native menu and save dialog, then produce an installable application.

The important result is not the note editor. It is the boundary around it: untrusted interface data crosses a narrow API before privileged code touches the computer.

Decide if Electron fits the app

Compare Electron with a website and native development before committing to a desktop application.

Electron is useful when the product needs desktop capabilities and the team can reuse web skills.

Start from the requirement, not the framework:

Requirement	First option to consider
Share by URL with no installation	Website
Local files, native menus, offline desktop workflow	Electron
Smallest download or deepest platform integration	Native app
Existing web product with a focused desktop companion	Electron or a thin native shell

Electron removes the need to build three separate native interfaces. It does not remove platform work. Menus, paths, installers, signing, permissions, and update behavior still differ.

For Desktop Notes, write a short decision record:

```
Desktop-only needs: offline notes, native export, application menu
Supported platforms: macOS and Windows
Accepted costs: larger download, two release pipelines
Reason a website is insufficient: notes must work fully offline
```

Make the tradeoff measurable. Set a download-size expectation, list the operating systems you will test, and name who owns security updates.

If the real app only displays a remote website, Electron may add more release and security responsibility than value. Choose it because desktop behavior matters.

Prepare the toolchain

Install a current Node.js release, Git, and the operating-system tools needed to build Electron applications.

Electron Forge creates and packages the project. Its current ecosystem tooling uses Node.js 22 as the minimum supported baseline.

Check both commands:

```
node --version
npm --version
git --version
```

Use the current Node.js LTS release. Your development Node.js runs npm, Forge, and build scripts. The packaged Electron app uses the Node.js version bundled inside Electron.

Those versions can differ. We will inspect the packaged runtime later with `process.versions` instead of assuming it matches `node --version`.

Packaging also uses operating-system tools. On macOS, install the Xcode command-line tools. On Windows, use PowerShell or Command Prompt for the course commands. On Linux, install the build packages reported by your distribution if a maker requests them.

Native Node.js modules add another boundary. They must be rebuilt for Electron's application binary interface. Forge handles common rebuild steps, but a module that works in plain Node.js can still fail inside Electron.

Create a normal development folder. Do not build the project inside a cloud-synchronized or network-mounted directory. Packaging tools create many files and work best on a local disk.

Record the versions in a small `TESTING.md` file. When packaging later fails on one machine, this baseline will tell you whether the toolchain changed.

Create the Electron Forge project

Scaffold a JavaScript Electron application with Forge and its Webpack template, then run the generated desktop window.

Electron Forge provides development, packaging, installer, and publishing steps in one project.

Create the application with the Webpack template:

```
npx create-electron-app@latest desktop-notes --template=webpack
cd desktop-notes
npm start
```

A small Electron window should open. Keep the terminal running while you work. Forge rebuilds the app when source files change.

The `@latest` tag asks npm for the current project generator. The generated `package.json` records the exact Electron and Forge versions used by the project.

Inspect those versions:

```
npm ls electron @electron-forge/cli
```

Keep the generated lockfile. The generator version is not enough to reproduce the project when dependency ranges resolve differently later.

Log the embedded runtime from `src/index.js`:

```
console.log(process.versions)
```

Compare it with `node --version` from the terminal. They can differ because Electron ships its own Node.js runtime.

In renderer DevTools, `typeof require` should be `'undefined'`. That difference is our first visible process boundary.

Commit the generated project before changing it. This gives you a clean point to compare when an experiment goes wrong.

Stop and restart `npm start` once. Hot rebuilding is useful, but lifecycle and preload changes often need a full app restart before you can trust the result.

Read the generated project

Identify the main, preload, renderer, and Forge configuration files before changing the application.

Before editing, map each generated file to the process that owns it:

- `src/index.js` starts the main process and creates the window.
- `src/preload.js` defines the narrow bridge available to the page.
- `src/renderer.js` runs with the interface.
- `src/index.html` contains the page.
- `forge.config.js` controls packaging and makers.

Open `package.json` too. The `main` field points at Forge output, while the scripts run Forge commands.

Trace startup in this order:

```
npm start
```

→ Electron Forge configuration
→ bundled main entry
→ BrowserWindow creation
→ preload bundle
→ HTML and renderer bundle

The Forge Webpack plugin supplies entry constants such as `MAIN_WINDOW_WEBPACK_ENTRY` and `MAIN_WINDOW_PRELOAD_WEBPACK_ENTRY`. They resolve differently in development and packaged builds. Use them instead of hardcoding a development-server URL or output path.

Do not rename files yet. First follow each entry from `package.json` to `forge.config.js`, then into `src/`.

Add a temporary log to `src/index.js` and another to `src/renderer.js`. Run the app and find each message. Main-process output appears in the terminal. Renderer output appears in Chromium DevTools.

Remove both logs. Knowing which process owns a file is the most important Electron debugging skill because errors, APIs, and trust differ at every boundary.

Check your understanding: foundations and setup

Check Electron tradeoffs, Forge, the generated project, required tools, and the roles of the first application files.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Windows and the renderer

Create a secure window and build the notes interface with browser JavaScript.

Understand the process model

Separate main-process responsibilities from renderer work before adding application behavior.

An Electron app is not one JavaScript environment.

The **main process** is the application coordinator. It owns lifecycle, creates windows, calls native APIs, and has Node.js access. If it blocks or crashes, the whole application is affected.

Each `BrowserWindow` loads web content in a **renderer process**. The renderer owns DOM events, interface state, and drawing. Treat its input as you would treat input from a web page.

The **preload script** runs in the renderer process before page code. With context isolation, it has a separate JavaScript world and can expose a narrow API through `contextBridge`.

Desktop Notes will follow this ownership map:

> **Diagram: Electron process ownership.** The renderer calls a narrow preload API, which sends validated actions to the main process. The main process returns note data and native menu events through the same bridge. The arrow is a trust boundary. Values are copied across IPC; the processes do not share ordinary objects or a call stack.

Keep CPU-heavy work out of the main process too. For this small app, file operations are asynchronous and short. Larger computation can move to a utility process or worker rather than freezing every window.

Before implementing a feature, write down which process owns it and why. If the answer is "the renderer because it was convenient," inspect whether the feature touches the computer or private data.

Create the main window

Configure a BrowserWindow with a preload script and load the Webpack renderer entry produced by Electron Forge.

Create windows in the main process after Electron is ready. Replace the generated window function in `src/index.js`:

```
const { app, BrowserWindow } = require('electron')

let mainWindow = null

function createWindow() {
  const win = new BrowserWindow({
    width: 960,
    height: 700,
    minWidth: 720,
    minHeight: 500,
    show: false,
    webPreferences: {
      preload: MAIN_WINDOW_PRELOAD_WEBPACK_ENTRY,
      contextIsolation: true,
      nodeIntegration: false,
      sandbox: true
    }
  })

  mainWindow = win
  win.on('closed', () => {
    if (mainWindow === win) mainWindow = null
  })

  win.once('ready-to-show', () => win.show())
  win.loadURL(MAIN_WINDOW_WEBPACK_ENTRY)

  return win
}
```

The two uppercase entry variables are supplied by the Forge Webpack plugin. One points to the renderer page and one points to the bundled preload script.

`show: false` avoids displaying a partially rendered window. `ready-to-show` displays it after the first useful paint. For an interface that renders slowly, a matching `backgroundColor` can be a better choice than delaying the whole window.

The security options are modern defaults, but keep them explicit:

- `nodeIntegration: false` keeps Node.js out of page code
- `contextIsolation: true` separates preload and page globals
- `sandbox: true` restricts the renderer process

Do not solve a preload import error by disabling one of these settings. A sandboxed preload has a limited environment, so move privileged Node.js work into the main process and expose one IPC method.

Open DevTools and verify `require` is not available to the page. The window should still load because the renderer bundle contains browser-compatible code.

Handle the window lifecycle

Follow macOS, Windows, and Linux conventions when the app starts, closes its windows, and becomes active again.

Electron APIs such as `BrowserWindow` are not ready when the module first loads. Create the first window after `app.whenReady()` resolves:

```
app.whenReady().then(() => {
  createWindow()

  app.on('activate', () => {
    if (BrowserWindow.getAllWindows().length === 0) createWindow()
  })
})

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

On Windows and Linux, closing every window normally quits the app. On macOS, an application usually remains active and creates a new window when its Dock icon is selected.

These are different lifecycle events:

- closing a window destroys that window
- `window-all-closed` means no application windows remain
- quitting ends the main process
- `activate` means the operating system brought the app forward

Do not keep using a destroyed `BrowserWindow` reference. Before sending a menu event, check that the target exists and has not been destroyed.

Test the complete sequence on every supported platform:

1. Launch the app and create one window.
2. Close that window.
3. On macOS, select the Dock icon and confirm a new window appears.
4. On Windows or Linux, confirm the application exits.
5. Quit from the menu and confirm no background process remains.

If the real app has unsaved changes, decide where close confirmation lives. Test both cancellation

and confirmed quit; lifecycle code that loses data is not complete.

Build the notes interface

Create the semantic HTML shell for a note list, editor form, and status message in the renderer.

The renderer is a web page, so start with ordinary accessible HTML. Replace `src/index.html` with a small two-column interface:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta
      http-equiv="Content-Security-Policy"
      content="default-src 'self'; script-src 'self'; style-src 'self'"
    />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Desktop Notes</title>
  </head>
  <body>
    <main class="app-shell">
      <aside>
        <button id="new-note" type="button">New note</button>
        <ul id="notes"></ul>
      </aside>
      <form id="editor">
        <input id="title" aria-label="Title" required />
        <textarea id="body" aria-label="Note" rows="16"></textarea>
        <button type="submit">Save note</button>
      </form>
    </main>
    <p id="status" role="status"></p>
  </body>
</html>
```

The Webpack template injects the renderer bundle, so keep the generated script integration instead if your current template includes an explicit entry.

The form works with a keyboard and the status message can announce save results. Add visible labels if the design later hides the purpose of either field; `aria-label` is not a substitute for clear visual text.

The content security policy allows only local scripts and styles. That means no inline `<script>`, inline event attribute, or remote stylesheet. Keep renderer code in the bundled entry instead of weakening `script-src` to fix an error.

At this point the interface has no desktop authority. Buttons can change DOM state, but they cannot read a file or open a native dialog. That is the boundary we want.

Use the keyboard to reach every control. Submit an empty title and confirm the browser's required-field behavior appears before adding custom validation.

Render notes safely

Keep note data in renderer state and draw user text without turning it into executable HTML.

Renderer state can be ordinary browser JavaScript. Start `src/renderer.js` with local state:

```
let notes = []
let selectedId = null

const list = document.querySelector('#notes')
const titleInput = document.querySelector('#title')
const bodyInput = document.querySelector('#body')
```

Render each title with `textContent`:

```
function renderNotes() {
  list.replaceChildren()

  for (const note of notes) {
    const button = document.createElement('button')
    button.type = 'button'
    button.textContent = note.title || 'Untitled'
    button.addEventListener('click', () => selectNote(note.id))

    const item = document.createElement('li')
    item.append(button)
    list.append(item)
  }
}
```

Do not place note content into `innerHTML`. Notes are user data, not markup. `textContent` keeps a note such as `<img onerror=...>` as visible text.

The code creates DOM nodes instead of building an HTML string. This keeps escaping decisions local and lets each button receive one explicit listener.

Use stable note IDs for selection. An array position changes when a note is inserted or deleted, so it is not a durable identity.

Test hostile-looking content now:

```
notes = [{
  id: 'test-note',
  title: '<img src=x onerror=alert(1)>',
  body: '<script>alert(1)</script>'
}]
```

The strings should appear as text. No request, alert, or script execution should occur. Keep this case in the renderer test checklist because later refactors can reintroduce unsafe HTML.

Rendering safely is separate from validating size and shape. The main process will still validate every note before saving it.

Check your understanding: windows and renderer

Check main and renderer responsibilities, BrowserWindow settings, application lifecycle, accessible HTML, and safe text rendering.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Preload and IPC

Expose narrow desktop capabilities and validate every privileged request.

Why the renderer is isolated

Understand why Node.js stays out of the page and why a preload bridge exposes only deliberate desktop capabilities.

Browser JavaScript normally cannot read arbitrary files or launch local programs. That limit contains the damage from an injected script or dependency bug.

Electron adds desktop power, but page code should not receive all of it. Keep three protections in place:

- `nodeIntegration: false` removes direct Node.js access from the page
- `contextIsolation: true` gives preload and page code separate globals
- `sandbox: true` restricts the renderer process

The preload script is more privileged than the page, even in a sandboxed renderer. Treat every value it exposes as a permanent public API.

This bridge is too powerful:

```
contextBridge.exposeInMainWorld('electron', {
  send: (channel, value) => ipcRenderer.send(channel, value)
})
```

Any renderer code can choose any channel. Future main-process handlers can accidentally become reachable without changing the bridge.

Expose one method per intention instead:

```
contextBridge.exposeInMainWorld('notesAPI', {
  saveNotes: notes => ipcRenderer.invoke('notes:save', notes)
})
```

This still does not make `notes` trusted. The main process must validate the sender and value before writing.

Imagine an injected script running in the renderer. List everything it could do through `window.notesAPI`. That list should be short, understandable, and no more powerful than the interface needs.

Expose a preload API

Use contextBridge to publish one renderer method for each allowed notes operation.

The preload script translates renderer intentions into fixed IPC channels. Replace `src/preload.js` with this bridge:

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

contextBridge.exposeInMainWorld('notesAPI', {
  loadNotes: () => ipcRenderer.invoke('notes:load'),
  saveNotes: notes => ipcRenderer.invoke('notes:save', notes),
  exportNotes: () => ipcRenderer.invoke('notes:export'),
  onNewNote: callback =>
    ipcRenderer.on('notes:new', () => callback())
})
```

The renderer will receive `window.notesAPI`. It does not receive the `ipcRenderer` object.

Notice how the event wrapper drops Electron's event object before calling renderer code. The callback receives no privileged sender reference.

Do not expose `ipcRenderer` directly. Current Electron releases prevent transferring the complete object over `contextBridge`, and a generic wrapper would recreate the same security problem.

Bridge values are copied or proxied across isolated JavaScript worlds. Pass plain serializable data. DOM nodes, Electron objects, and functions nested inside note data cannot cross IPC as application records.

Each method name describes one capability. This API is easy to audit and can stay stable even if the IPC channel names change.

Open the renderer console and inspect `window.notesAPI`. Confirm that `loadNotes` exists while `require`, raw channel selection, and filesystem methods do not.

For a long-lived page that registers listeners repeatedly, add a matching unsubscribe operation. Desktop Notes registers `onNewNote` once during startup, so one listener matches the page lifetime.

Handle two-way IPC

Pair ipcRenderer.invoke with ipcMain.handle when the renderer needs a result from the main process.

Loading notes is a request with one asynchronous response. Pair `ipcRenderer.invoke()` with `ipcMain.handle()`.

Register each handler once in the main process:

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

ipcMain.handle('notes:load', event => {
  assertTrustedSender(event.senderFrame)
  return []
})

app.whenReady().then(() => {
  createWindow()
})
```

Call the handler from the renderer:

```
async function start() {
  notes = await window.notesAPI.loadNotes()
  renderNotes()
}

start()
```

`invoke()` returns a promise. The value returned by the main handler becomes the resolved renderer value.

The main process must validate the sender before returning data or performing privileged work. We will add data validation next.

Use one channel for one request shape. Do not create a generic `notes:command` handler with an operation name and arbitrary payload. Separate handlers make validation, authorization, and error behavior visible.

Prefer this asynchronous pattern to synchronous IPC. A synchronous renderer request blocks the renderer while the main process works and can freeze both sides through dependency cycles.

Thrown handler errors reject the renderer promise, but Electron does not preserve arbitrary error details across IPC. Return user-safe messages and keep internal diagnostics in the main process.

Call `loadNotes()` twice and inspect the main-process log. Each invocation should reach the same one registered handler, not duplicate listeners created every time a window opens.

Validate IPC input

Check the sender and every note field in the main process before writing renderer data to disk.

A TypeScript type or renderer form check cannot protect the main process at runtime. Renderer values are untrusted when they cross IPC.

Add a small validator:

```
function validateNotes(value) {
  if (!Array.isArray(value)) throw new Error('Notes must be an array')
  if (value.length > 1000) throw new Error('Too many notes')

  return value.map(note => {
    if (!note || typeof note !== 'object') throw new Error('Invalid note')
    if (typeof note.id !== 'string' || note.id.length > 100) {
      throw new Error('Invalid note id')
    }
    if (typeof note.title !== 'string' || note.title.length > 200) {
      throw new Error('Invalid note title')
    }
    if (typeof note.body !== 'string' || note.body.length > 100000) {
      throw new Error('Invalid note body')
    }

    return { id: note.id, title: note.title, body: note.body }
```

```
  })
}
```

The validator returns a new object with only allowed properties. That drops prototype tricks and unexpected fields instead of passing the renderer object into storage.

Validate the sender separately. For one known main window:

```
function assertTrustedSender(frame) {
  if (!mainWindow || frame !== mainWindow.webContents.mainFrame) {
    throw new Error('Untrusted IPC sender')
  }
}
```

Checking only a URL string is easy to get wrong. Comparing the actual main frame is a strong fit when this app has one trusted local window. If the app later adds several windows, define an explicit allowlist of trusted frames and capabilities.

Return plain data. Electron serializes IPC values with the structured clone algorithm, so DOM elements and Electron objects cannot cross the boundary.

Test the validator directly with valid notes, an object instead of an array, 1,001 notes, oversized strings, extra properties, and an object with a custom prototype. Then send a valid value from an iframe and confirm the sender check rejects it.

Validation limits shape and volume. It does not decide which user may save which notes. Add authorization too when data belongs to multiple users.

Finish renderer editing

Create, select, update, and delete notes in renderer state, then save through the preload API.

Keep selection, form input, and unsaved state in the renderer. Only privileged persistence crosses the bridge.

Create a new note with a browser-generated ID:

```
function createNote() {
  const note = {
    id: crypto.randomUUID(),
    title: 'Untitled',
    body: ''
  }

  notes.unshift(note)
  selectedId = note.id
  renderNotes()
  showSelectedNote()
}
```

When the form submits, copy the current values into the selected note. Then await the save:

```
async function save() {
  status.textContent = 'Saving…'
```

```
  try {
    const result = await window.notesAPI.saveNotes(notes)
    status.textContent = `Saved ${result.saved} notes`
  } catch {
    status.textContent = 'Could not save notes'
  }
}
```

Do not clear the editor on failure. The unsaved note must remain visible so the user can retry or copy it.

For deletion, remove only the selected ID, select the next available note, render, and save again.

Connect `window.notesAPI.onNewNote(createNote)` too. A native menu will call that renderer action later.

Think through overlapping saves. Disable the Save button while one write is pending, or serialize writes so an older slow request cannot finish after a newer one and overwrite it.

Exercise the failure path by temporarily making the main handler throw. Confirm the status changes, the note remains editable, and no success message appears before the promise resolves.

Check your understanding: preload and IPC

Check context isolation, narrow bridge methods, invoke and handle, sender validation, input validation, and IPC serialization.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Local data and native APIs

Persist notes and add native dialogs, menus, and keyboard shortcuts.

Choose the data location

Store application data under Electron's per-user userData directory instead of beside source files or the executable.

A packaged application can live in a read-only folder. Its current working directory and source layout can also differ between development and production.

Electron provides platform-specific locations through `app.getPath()`. Use `userData` for small configuration and application data.

Create a dedicated subdirectory inside it:

```
const path = require('node:path')

function notesPath() {
  return path.join(
    app.getPath('userData'),
    'desktop-notes-data',
```

```
    'notes.json'
  )
}
```

The actual path differs on macOS, Windows, and Linux. Let Electron choose it instead of assembling a home-directory path yourself.

Call `app.getPath()` only after Electron is ready. Application identity affects the default directory, so choose a stable `productName` before shipping.

Do not put Chromium caches beside durable notes. Electron exposes `sessionData` for cookies, local storage, network state, and disk cache. Those files can grow large and have a different cleanup policy.

Log the chosen path during development:

```
console.log(notesPath())
```

Open that folder, create a note, and confirm only your data file appears in the dedicated subdirectory. Remove the log before release if paths are considered sensitive in your support model.

Decide what uninstall means. Operating systems often leave `userData` behind, which preserves notes across reinstalls but may surprise users who expected removal.

Read and write the notes file

Create the data directory, handle a missing file, validate stored JSON, and write notes asynchronously.

Create `src/notes-store.js`. Keep file operations outside IPC registration so we can test them without Electron.

```
const fs = require('node:fs/promises')
const path = require('node:path')

async function readNotes(filePath, validateNotes) {
  try {
    const contents = await fs.readFile(filePath, 'utf8')
    return validateNotes(JSON.parse(contents))
  } catch (error) {
    if (error.code === 'ENOENT') return []
    throw error
  }
}

async function writeNotes(filePath, notes) {
  await fs.mkdir(path.dirname(filePath), { recursive: true })
  const temporaryPath = `${filePath}.tmp`

  await fs.writeFile(
    temporaryPath,
    JSON.stringify(notes, null, 2),
    'utf8'
  )
```

```
  await fs.rename(temporaryPath, filePath)
}

module.exports = { readNotes, writeNotes }
```

A missing file means the app has no notes yet. Invalid JSON is different: surface the error instead of silently erasing or replacing user data.

The asynchronous file APIs keep the main process responsive while the operating system works.

Writing a temporary file first reduces the chance of leaving a half-written JSON document when the process stops during a save. The rename replaces the visible file only after the complete temporary file exists.

This is not a complete backup system. Disk errors, external edits, and filesystem behavior can still fail. Keep the original error, preserve the unsaved renderer state, and consider versioned backups for important production data.

JSON parsing and data validation are separate steps. Valid JSON can still contain an invalid note shape, so `readNotes()` passes the parsed value through `validateNotes()` before returning it.

Test three files by hand: no file, valid notes, and truncated JSON. Only the missing file should become an empty list.

Connect persistence to IPC

Wire validated load and save handlers to the notes store and report failures without leaking internal details.

Connect the store to the main-process handlers in `src/index.js`:

```
const { readNotes, writeNotes } = require('./notes-store')

ipcMain.handle('notes:load', async event => {
  assertTrustedSender(event.senderFrame)
  return readNotes(notesPath(), validateNotes)
})

ipcMain.handle('notes:save', async (event, value) => {
  assertTrustedSender(event.senderFrame)
  const notes = validateNotes(value)
  await writeNotes(notesPath(), notes)
  return { saved: notes.length }
})
```

The load path validates data read from disk. The save path validates renderer data before opening a file. Neither side is trusted because local files can be corrupted or edited outside the app.

Register these handlers once, not every time a window opens. If you replace a handler during a development reload, remove the old handler deliberately before adding another.

In the renderer, wrap bridge calls in `try` and `catch`. Show “Could not save notes” to the user. Keep detailed paths and stack traces in main-process diagnostics, not in the renderer message.

Do not claim a save succeeded before the promise resolves. If the write fails, keep the unsaved note

visible so the user can copy it.

Run this persistence exercise:

1. Create a note and wait for the saved status.
2. Quit the whole application, then reopen it.
3. Confirm the note returns from the expected `userData` file.
4. Make the data folder temporarily unwritable and attempt another save.
5. Confirm the renderer reports failure without losing the edited text.

Restore the folder permissions after the test. A successful write plus a successful restart proves much more than seeing the object in renderer memory.

Export with a native dialog

Use Electron's save dialog in the main process and write a user-selected JSON export file.

The renderer requests an export, but the main process owns both the native dialog and filesystem write.

Add this IPC handler:

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

ipcMain.handle('notes:export', async event => {
  assertTrustedSender(event.senderFrame)
  const notes = await readNotes(notesPath(), validateNotes)
  const result = await dialog.showSaveDialog(mainWindow, {
    defaultPath: 'desktop-notes.json',
    filters: [{ name: 'JSON', extensions: ['json'] }]
  })

  if (result.canceled || !result.filePath) return { canceled: true }

  await writeNotes(result.filePath, notes)
  return { canceled: false }
})
```

Passing `mainWindow` makes the dialog modal to the notes window. The operating system returns either a user-selected path or a canceled result.

Do not let the renderer provide an arbitrary export path. That would turn a narrow export feature into a general filesystem write primitive.

Read and validate the current store before exporting. This makes the exported file reflect durable notes, not a renderer value that failed validation or has not been saved.

Add an Export button that calls `window.notesAPI.exportNotes()` and reports whether the operation completed or was canceled.

Treat cancellation as a normal outcome, not an error. Test cancel, overwrite confirmation, an unwritable destination, and successful export. Open the resulting JSON with another program and confirm its contents.

For sensitive notes, remember that export creates a second unmanaged copy. Document that be-

havior and never upload it silently.

Add an application menu

Create a cross-platform menu with native roles and send a narrow New Note event to the renderer.

Electron menus use templates. Built-in roles follow operating-system labels, placement, and common keyboard behavior.

```
const { Menu } = require('electron')

function installMenu(win) {
  const template = [
    ...(process.platform === 'darwin' ? [{ role: 'appMenu' }] : []),
    {
      label: 'File',
      submenu: [
        {
          label: 'New Note',
          accelerator: 'CmdOrCtrl+N',
          click: () => {
            if (!win.isDestroyed()) {
              win.webContents.send('notes:new')
            }
          }
        },
        { role: 'quit' }
      ]
    },
    { role: 'editMenu' },
    { role: 'viewMenu' },
    { role: 'windowMenu' }
  ]

  Menu.setApplicationMenu(Menu.buildFromTemplate(template))
}
```

Call `installMenu(win)` after creating the window.

The menu runs in the main process. It sends one fixed event to the known renderer, and preload removes Electron's event object before calling `createNote()`.

The `appMenu` role provides the conventional first macOS menu. Other roles provide platform shortcuts and labels better than handwritten copies.

Test the menu with a mouse and `CmdOrCtrl+N`. Close the window and trigger the command if the platform still exposes the menu. The destroyed-window guard should prevent a send to dead web contents.

If a menu action performs privileged work, do it in the main process or call the same validated function as IPC. Do not create a second unchecked implementation just for the menu.

Check your understanding: local data and native APIs

Check userData paths, JSON failure handling, awaited persistence, native save dialogs, application menus, and platform conventions.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Security and quality

Restrict renderer authority, test data behavior, debug, and update safely.

Keep secure window defaults

Review the BrowserWindow boundary and understand what context isolation, sandboxing, and disabled Node integration protect.

Every window is a separate security decision. Return to each `BrowserWindow` and verify these settings:

```
webPreferences: {
  preload: MAIN_WINDOW_PRELOAD_WEBPACK_ENTRY,
  contextIsolation: true,
  nodeIntegration: false,
  sandbox: true
}
```

Modern Electron releases default to context isolation and renderer sandboxing. Do not disable either feature to make an import work.

If renderer code needs an npm package, bundle a browser-compatible package through Webpack. If it needs files or native Electron APIs, add one validated preload capability.

Never load remote content into a window that has Node integration. Remote content can change outside your release process.

Also confirm that you did not weaken adjacent controls:

- `webSecurity` remains enabled
- insecure mixed content is not allowed
- experimental Blink features are not enabled casually
- the page has a restrictive Content Security Policy
- navigation, new windows, and permissions are explicitly controlled

Defaults can change and generated configuration can hide an override. Keeping the important values explicit makes code review easier.

Run a boundary test in renderer DevTools. `require('node:fs')` should not work, while `window.notesAPI.loadNotes` should exist. Then inspect the preload API and verify it exposes no raw Electron object.

Security options reduce damage; they do not make renderer input trusted. IPC sender checks and value validation remain necessary.

Restrict navigation and new windows

Prevent the notes window from navigating away or creating an unexpected BrowserWindow.

Desktop Notes loads one bundled interface. It never needs to show a different page, so any navigation is by definition an attack or a bug.

The threat is concrete: if injected markup ever reaches the renderer — a crafted note, a templating slip — navigation is how it escalates. A window that navigates to an attacker's page hands that page your preload bridge and your app's identity. Renderer-created windows are the same problem through a second door: `window.open()` and `target="_blank"` spawn new pages you never planned for.

Desktop Notes does not need page navigation or renderer-created windows, so deny both:

```
win.webContents.on('will-navigate', event => {
  event.preventDefault()
})

win.webContents.setWindowOpenHandler(() => ({
  action: 'deny'
}))
```

`will-navigate` fires when the top frame is about to load a new URL — a clicked link, a form submission, or a `location` assignment. Calling `event.preventDefault()` cancels it. The window open handler runs before any new window exists, and returning `{ action: 'deny' }` means one is never created. It does not fire for in-page anchor jumps, so nothing legitimate breaks.

Allowing one external link

If you later add an external documentation link, intercept it in the main process:

```
const { shell } = require('electron')

win.webContents.setWindowOpenHandler(({ url }) => {
  const target = new URL(url)

  if (
    target.protocol === 'https:' &&
    target.hostname === 'docs.desktop-notes.test'
  ) {
    void shell.openExternal(target.href)
  }

  return { action: 'deny' }
})
```

The allowed link opens in the user's browser, and the answer inside the app stays deny — always.

Parse the URL and compare exact protocol and hostname values. A `startsWith()` check can accept attacker-controlled hosts that merely begin with trusted text.

Never pass an unchecked string to `shell.openExternal()`. Protocol handlers can do more than open web pages.

Keep the restrictive content security policy in `index.html`. Avoid inline scripts because they require weakening `script-src`.

Verify the lockdown

Run the app, open DevTools, and try to escape:

```
window.open('https://attacker.invalid')
location.href = 'https://attacker.invalid'
```

Nothing should open, and the notes interface should stay put. Then test a normal click, `window.open()`, a link with `target="_blank"`, and a form navigation. Include confusing URLs such as `https://docs.desktop-notes.test.attacker.invalid` and a non-HTTP protocol. Only the exact allowlisted link should leave the app — and it should leave through the system browser, never inside your window.

Deny unused permissions

Install a permission handler that rejects camera, microphone, notification, and other web permissions the app does not use.

Chromium exposes permissions for media, geolocation, screen capture, notifications, devices, and filesystem features. Desktop Notes needs none.

Install both permission handlers after the app is ready:

```
const { session } = require('electron')

session.defaultSession.setPermissionRequestHandler(
  (_webContents, _permission, callback) => {
    callback(false)
  }
)

session.defaultSession.setPermissionCheckHandler(() => false)
```

Electron documents both handlers because web APIs can check permission before requesting it. Implementing only one leaves an incomplete policy.

If the app later needs one permission, check the exact permission, requesting origin, embedding origin, and frame details. Deny everything else.

A prompt is not a security policy. The handler should decide which requests are valid before Chromium presents or grants them.

Exercise the policy from renderer DevTools with a harmless permission request, such as notifications. Confirm it is denied and no operating-system prompt appears.

If you add a custom session or partition later, install the same policy there. `defaultSession` does not configure unrelated sessions automatically.

Test the store and validator

Move data rules into plain modules and test valid notes, malformed data, missing files, and round-trip persistence with Node.js.

The easiest Electron code to test is code that does not depend on Electron. This is why storage and validation live in plain modules.

Export `validateNotes()`, `readNotes()`, and `writeNotes()`. Write Node tests around temporary files:

```
const test = require('node:test')
const assert = require('node:assert/strict')
const fs = require('node:fs/promises')
const os = require('node:os')
const path = require('node:path')

test('notes survive a file round trip', async () => {
  const folder = await fs.mkdtemp(path.join(os.tmpdir(), 'notes-'))
  const file = path.join(folder, 'notes.json')
  const notes = [{ id: '1', title: 'Trip', body: 'Pack light' }]

  try {
    await writeNotes(file, notes)
    assert.deepEqual(await readNotes(file, validateNotes), notes)
  } finally {
    await fs.rm(folder, { recursive: true, force: true })
  }
})
```

Add tests for a missing file, invalid JSON, more than 1,000 notes, an oversized title, and unexpected properties.

Assert the failure, not only that "something threw":

```
assert.throws(
  () => validateNotes([{ id: '1', title: 'x'.repeat(201), body: '' }]),
  /Invalid note title/
)
```

Add a write-failure test with an invalid or unwritable destination appropriate to the test platform. Confirm the original notes file remains readable after a failed replacement.

Keep one manual checklist for window lifecycle, menu shortcuts, save dialog cancellation, and renderer error messages.

The remaining boundary needs an integration test through Electron: a renderer invokes IPC, main validates the sender and payload, and the bridge returns a plain result. Run that separately from fast Node tests.

Debug and update deliberately

Use the correct DevTools, inspect main-process output, and keep Electron current without mixing dependency upgrades with feature work.

Start diagnosis in the process that owns the failing code. Renderer and preload messages appear in Chromium DevTools. Main-process failures appear in the terminal that ran `npm start`.

Open renderer tools during development only:

```
if (!app.isPackaged) {
  win.webContents.openDevTools({ mode: 'detach' })
}
```

When an IPC request fails, check both places. The renderer may show a rejected promise while the main process contains the original error.

Use this sequence:

1. Reproduce one action from a clean launch.
2. Inspect the renderer rejection and Network or Console details.
3. Find the matching main-process handler and diagnostic.
4. Verify the sender check, input validator, filesystem path, and returned value.
5. Repeat in the packaged app, where paths and bundles differ.

Observe process failures explicitly during development:

```
win.on('unresponsive', () => {
  console.error('Main window became unresponsive')
})

app.on('render-process-gone', (_event, contents, details) => {
  console.error('Renderer exited', contents.id, details.reason)
})
```

Do not dump note bodies or credentials while debugging. Log safe IDs, operation names, and error categories.

Keep Electron on a supported release. Electron includes Chromium and Node.js, so framework updates also deliver security fixes from those projects.

Upgrade one Electron major version at a time. Read the matching versioned documentation and breaking changes. Run Node tests, renderer/IPC checks, packaged-app checks, and the security checklist before changing other dependencies.

Record `process.versions.electron`, `process.versions.chrome`, and `process.versions.node` from the main process after each upgrade. This proves which runtime the packaged application actually uses.

Check your understanding: security and quality

Check window isolation, navigation controls, permissions, external links, test boundaries, process-specific debugging, and Electron updates.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Package and ship

Create distributables, add platform assets, sign releases, and plan updates.

Set application metadata

Give the package a stable name, version, description, author, and identifier before producing distributable files.

Application identity reaches the operating system, update service, signing system, and data directory. Set it before the first public build.

Open `package.json` and replace the generated placeholders.

Use a package-safe `name`, a human-readable `productName`, a semantic `version`, and accurate author and description fields.

Set a stable application identifier in `forge.config.js` through the packager configuration:

```
module.exports = {
  packagerConfig: {
    name: 'Desktop Notes',
    appBundleId: 'com.flaviocopes.desktopnotes'
  },
  makers: [
    // generated makers stay here
  ]
}
```

Choose the identifier once. Signing, updates, operating-system permissions, and user data can depend on application identity.

Use one identity map:

Field	Purpose
`name`	Package-safe internal name
`productName`	Human-readable application name
`version`	Release version
`appBundleId`	Stable macOS application identifier
Windows application identity	Installer and upgrade continuity

The exact Windows identifier depends on the maker. Configure it in that maker instead of assuming `appBundleId` covers every platform.

Build once, then inspect the packaged application name, executable, metadata, user-data location, and installer identity. They should all describe Desktop Notes consistently.

Increment the version for every release. Never replace an old artifact with different code under the same version; update clients and support logs need an immutable version-to-code mapping.

Package and make distributables

Use Electron Forge to create an unpacked application first, then build the platform-specific installer or archive.

Forge separates two useful failure boundaries.

Packaging builds the application directory. **Making** wraps that packaged app in a distributable format.

Create an unpacked application:

`npm run package`

Open the generated app under `out/` and repeat the core manual checks. This catches paths that worked only through the development server.

Stop `npm start` before testing. The packaged app must not depend on a live Webpack development server or source file outside its bundle.

Check load, edit, save, restart, export, menu shortcuts, navigation denial, and permission denial. Inspect the `userData` location to confirm packaged and development identities behave as intended.

Then create a distributable:

`npm run make`

A **maker** converts the packaged app into a platform format such as a DMG, ZIP, Squirrel installer, DEB, or RPM.

Makers are platform-specific. Build macOS artifacts on macOS, Windows artifacts on Windows, and Linux artifacts on the matching Linux environment unless the selected maker explicitly documents another setup.

When `make` fails, find the stage first. A renderer bundling error, native-module rebuild error, signing error, and maker error need different fixes.

Install the distributable on a clean test account. Launching the unpacked folder proves packaging; installing and uninstalling prove the maker result.

Add application icons

Prepare platform icon files and configure Forge so the packaged app no longer uses the default Electron icon.

Start with a square master image that has enough resolution and transparent padding. An icon must remain recognizable at launcher and notification sizes, not only at 1024 pixels.

Platforms expect different formats. macOS uses `.icns`, Windows uses `.ico`, and Linux packages commonly use PNG files at several sizes.

Set the icon base path in `packagerConfig`. Electron Packager adds the platform extension where appropriate:

```
packagerConfig: {
  icon: './assets/icon'
}
```

Some makers also need icon settings in their own configuration. Check the maker documentation for every target.

Keep generated platform assets in source control or document the exact generation command. A manually converted icon that exists only on one release machine makes the build irreproducible.

Delete old `out/` artifacts before checking a change. Operating systems and installers cache icons aggressively, so an old installed copy can make correct configuration look broken.

Package again and inspect:

- application bundle or executable
- Dock, taskbar, and task switcher
- installer or disk image
- installed shortcut and application list
- file metadata shown by the operating system

Test light and dark desktop themes where the platform uses them. Do not add a colored background merely to hide poor contrast at small sizes.

Sign the application

Understand why macOS and Windows warn about unsigned software and prepare platform credentials without committing secrets.

Code signing connects an artifact to a publisher identity and lets the operating system detect later changes.

macOS distribution normally needs a Developer ID certificate plus notarization. Windows distribution normally needs an Authenticode certificate. These are separate platform trust systems.

Signing configuration depends on the platform, certificate provider, and release environment. Follow the current Forge guide instead of copying old flags from a tutorial.

Keep certificate files, passwords, API keys, and signing tokens in the release system's secret store. Never add them to `forge.config.js`, `.env` files committed to Git, or build logs.

Sign release artifacts in a controlled CI job or dedicated release environment. Verify the signature on the final artifact, not only on an intermediate app folder.

On macOS, verify the signed bundle and the notarized result. On Windows, inspect the final executable or installer with the operating-system signature tools. Then download the published artifact and verify again; this checks that upload did not substitute a different file.

Signing proves publisher identity and integrity. It does not prove the application is harmless. Keep dependency review, malware scanning where required, least privilege, and runtime security controls.

Create a failure exercise in a test release: omit signing credentials and confirm the pipeline fails or clearly marks the artifact unsigned. A release job should never silently publish an artifact whose signing step was skipped.

Plan publishing and updates

Publish immutable artifacts, test each supported platform, and choose an update path before users depend on the app.

A release is an upgrade path from code you shipped before. Treat the artifact, metadata, signature, and update policy as one unit.

Build and test every supported operating system and architecture. Install over the previous version, launch from a clean account, create and export a note, restart, and uninstall.

Publish each artifact with its version, platform, architecture, and checksum. Keep old releases available long enough to diagnose regressions.

Electron Forge publishers upload artifacts; they do not automatically make every artifact safely updateable. Electron's built-in updater supports macOS and Windows, while Linux distribution

commonly follows the package manager or another platform-specific path.

Automatic updates need:

- immutable versioned artifacts and compatible update metadata
- HTTPS from a controlled update source
- signed applications, including the macOS signing prerequisite
- staged testing from the previous supported version
- a response plan for a bad release
- clear behavior when download, verification, or installation fails

Start with manual downloads if you are not ready to operate updates safely. Add automatic updates only after signed release artifacts and upgrade tests are routine.

Write a short release checklist now. Include versioning, tests, packaging, signing, malware scanning where applicable, upload, download verification, and announcement.

Add two update exercises. First, update from version `1.0.0` to `1.0.1` and confirm notes remain under the same application identity. Then make the update source unavailable and confirm the installed app still starts and preserves data.

Do not rely on "latest" as a rollback strategy. Once a broken update installs, you need a tested recovery release with a higher version or a documented manual recovery path.

Final check: package and ship

Check application identity, package and make, platform makers, icons, signing secrets, release testing, publishing, and updates.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/electron/.