Lab 3: Electron/Angular Testcase for macOS

YT0798 Cross Development, Thomas More

Gilles Van pellicom

December 16, 2025

Contents

Notes on the Process

Reproducibility

- **Development, Compilation & Testing environment:**
  - **Platform:** macOS Tahoe 26.2
  - **Hardware:** M3 Pro, 36 GB LPDDR5
  - **IDE:** IntelliJ 2025.3

Chapter 1

Testcase

1.1 Requirements and Fulfillment

The test case is an Electron port of the Lab 1 calculator test case. It meets the following criteria:

- **Dialog for selecting an accent color:** Settings dialog includes `<input type="color">` for accent color selection.
- **File I/O for settings storage:** Selected accent color stored persistently using Node.js API.
- **IPC call between main and render process:** `CalculatorService` and `SettingsService` are shared between pages `tab1`, `tab2` and settings modal. These services are located under `/src/app/services`
- **Squirrel installer:** npm generated package for stand-alone deployment.
- **Security:** Followed security best practices and hardened application.

1.2 Use of Node.js APIs and IPC Mechanisms

This project makes deliberate and constrained use of Node.js APIs, following Electron security best practices. All Node.js-specific functionality is confined to trusted execution contexts, namely the main process and the preload script, both of which run in a Node.js-capable environment. The renderer process is intentionally isolated from direct Node.js access.

1.2.1 Node.js API Usage in the Main Process

The main process, implemented in `src/electron-main.ts`, executes in a full Node.js environment and is responsible for system-level interactions. Several core Node.js modules and globals are used in this context.

File system operations are performed via the `fs` module. Specifically, `fs.readFileSync` is used to synchronously load application settings from persistent storage, while `fs.writeFileSync` serializes and writes updated settings back to disk. These operations are typical for configuration management in desktop applications.

Path manipulation is handled using the `path` module. Calls to `path.join` are used to construct platform-independent file system paths, ensuring compatibility across operating systems such as Windows, macOS, and Linux.

The application also relies on Node.js global objects for environment awareness. The `process.platform` property is queried to implement platform-specific behavior, for example when handling application lifecycle events. Additionally, the `__dirname` global is used in conjunction with `path.join` to resolve absolute paths relative to the location of the executing file.

1.2.2 Node.js API Usage in the Preload Script

The preload script, located at `src/preload.js`, runs in a privileged context that bridges the gap between the renderer and the main process. While it has access to browser APIs, it also retains controlled access to Node.js functionality.

In this file, the Node.js CommonJS module system is used via `require('electron')`. Although the imported module is provided by Electron, the `require` function itself is a fundamental Node.js API and illustrates that the preload script operates within a Node.js-enabled environment.

1.2.3 Renderer Process Isolation and IPC

In contrast, the renderer process (including `main.ts` and Angular components) does not have direct access to Node.js APIs. This restriction is enforced by disabling Node integration (`nodeIntegration: false`) and is a critical security measure.

Whenever the renderer requires functionality that depends on Node.js capabilities, such as file system access, it must communicate with the main process through Inter-Process Communication (IPC). These IPC channels are explicitly exposed by the preload script, ensuring a clear separation of concerns and minimizing the attack surface of the application.

1.3 Portability Considerations

Since the project was built in Electron, the project was built with portability from the ground up. On my development machine the project required no special considerations. Some highlights of using Electron for portability:

- **Node.js API via IPC:** Safe and controlled OS-API interaction via an OS-agnostic manner.
- **OS-agnostic color picker:** Leveraged Angular's `<input type="color">` to handle color selection, allowing Electron to manage platform-specific details seamlessly.
- **Full chromium browser and CSS support:** Transferable skill-set for people with web development experience.

Chapter 2

Security

This chapter outlines the security measures implemented in the Electron application to ensure safe inter-process communication (IPC) and protection against common vulnerabilities. The practices described follow the latest Electron security recommendations.

2.1 Electron Security Best Practices

The application follows the guidelines provided in the official Electron documentation. Key practices implemented are as follows:

1. **Only load secure content:** All content is loaded from the local filesystem using the `file://` protocol. No remote content is loaded. Navigation attempts to external URLs are explicitly blocked via the `will-navigate` event handler.
2. **Disable Node.js integration for remote content:** Node.js integration is disabled for all renderer processes.
3. **Enable context isolation:** `contextIsolation` is enabled for all renderers to prevent untrusted scripts from accessing Electron APIs.
4. **Enable process sandboxing:** All renderers run in a sandboxed environment (`sandbox: true` in `webPreferences`).
5. **Use `ses.setPermissionRequestHandler()` in sessions with remote content:** The application does not load remote content, so this is not applicable.
6. **Do not disable web security:** The default `webSecurity` setting is preserved.
7. **Define a Content Security Policy:** The application enforces a strict CSP, allowing scripts only from the application's origin (`script-src 'self'`), fully preventing `eval()` and `new Function()` usage.
8. **Do not enable `allowRunningInsecureContent`:** This option is not enabled; the default setting is preserved.
9. **Do not enable experimental features:** No experimental features are enabled.
10. **Do not use `enableBlinkFeatures`:** This option is not used.
11. **Do not use `<webview>` allowpopups:** The `<webview>` tag is not used.
12. **Verify `<webview>` options and parameters:** The `<webview>` tag is not used.
13. **Disable or limit navigation:** Navigation is restricted to `file://` URLs. Any attempt to navigate to an external URL is blocked.

14. **Disable or limit creation of new windows:** New window creation is disabled using `setWindowOpenHandler`.
15. **Do not use `shell.openExternal` with untrusted content:** `shell.openExternal` is not used.
16. **Use a current version of Electron:** The project is kept up-to-date with the latest Electron version to ensure all known vulnerabilities are patched.
17. **Validate the sender of all IPC messages:** IPC messages are validated to ensure that only windows with `file://` URLs can trigger IPC calls.
18. **Avoid usage of `file://` protocol when possible:** Currently, the application uses `file://` to load local content. A custom protocol could be used in the future for improved security.
19. **Check which Electron Fuses can be changed:** Electron Fuses could be used to further harden the application. This has not yet been implemented.
20. **Do not expose Electron APIs to untrusted web content:** Electron APIs are not exposed to untrusted content. Only a limited API is exposed to the renderer via a preload script and `contextBridge`.

2.2 Eval function

The javascript function used to evaluate mathematical expressions can be utilised for injection. Since the use of this function is not allowed by our Content Security Policy, a local evaluator was provided.

Chapter 3

Toolchain Setup and Workflow

3.1 Prerequisites

The Electron framework requires Node.js as a prerequisite. On macOS, Node.js was installed using the Homebrew package manager:

```
brew install node
```

Subsequently, Electron was installed globally via `npm`:

```
npm install -g electron
```

To verify that the installation was successful, the Electron version was checked:

```
electron --version
```

3.2 Project Initialization

A project directory was created and initialized with `npm` as follows:

```
mkdir my-electron-app
cd my-electron-app
npm init -y
```

Electron was then added as a development dependency:

```
npm install --save-dev electron
```

A start script was added to `package.json` to facilitate launching the application:

```
"scripts": {
  "start": "electron ."
}
```

The application was launched to confirm that the setup was functional:

```
npm start
```

3.3 Running the Electron Application

The project provides several `npm` scripts to facilitate development and execution of the Electron application. The primary scripts relevant to running the application are described below.

3.3.1 Development Mode

To run the application in development mode, the following script is used:

`npm run electron:dev`

This script performs several tasks concurrently:

1. `npm run watch` — Continuously rebuilds the Angular frontend in development mode.
2. `npm run build:electron` — Compiles the Electron main process using the TypeScript configuration defined in `tsconfig.electron.json`.
3. `wait-on file:./dist/index.html` — Waits until the frontend build is complete before starting Electron.
4. `npm run electron` — Launches the Electron application, using the compiled main process located at `build-electron/electron-main.js`.

This setup allows simultaneous rebuilding of the frontend and backend while keeping the Electron application in sync with the latest changes.

3.3.2 Manual Launch

For manual execution, the Electron application can also be started directly with:

`npm run electron`

This command launches the application using the already compiled Electron main process and assumes the frontend files are available in the `dist` directory.

3.4 Squirrel Installer and Packaging

3.4.1 Build Configuration

The project uses `electron-builder` to create distributable packages. For Windows, the target installer format is `Squirrel`, which allows automatic updates and easy installation. For macOS however, `Squirrel` is not used since this does not properly integrate with modern Apple-enforced macOS code signing, notarization and M-type silicon. The standard electron builder was used instead.

The relevant build configuration in `package.json` is as follows:

```
"build": {
  "appId": "com.crossdev3.app",
  "productName": "CrossDev3",
  "directories": { "output": "release" },
  "files": [
    "build-electron/**/*",
    "dist/**/*",
    "preload.js"
  ],
  "win": {
    "target": "squirrel",
    "icon": "build/icon.ico"
  },
  "squirrelWindows": {
    "iconUrl": "https://icon-icons.com/download-file?file=https%3A%2F%2Fimages.icon-icons.com%2F3053%2F
  }
}
```

This configuration specifies:

- `appId` and `productName` for identifying the application.
- `directories.output`, which sets the folder where packaged installers will be saved.
- Files and directories to include in the package (`build-electron`, `dist`, and `preload.js`).
- Windows-specific build target (`squirrel`) and icon.
- Icon URL for Squirrel updates.

3.4.2 Building the Installer

The installer is created using the following sequence of commands:

1. `npm run build` — Builds the Angular frontend.
2. `npm run build:electron` — Compiles the Electron main process.
3. `npm run package` — Invokes `electron-builder`, producing the installer files.

Upon completion, the Squirrel installer and other distributables can be found in the `/release` directory under the appropriate architecture folder.

3.5 Outcome

A working native-macOS build of the testcase was achieved utilising Electron/Angular, and was tested on device. The test method was a macOS .app file, which equates to a standard desktop app, since macOS doesn't utilise installers for regular apps. The installers (.pkg) files are reserved for more intrusive software packages such as antivirus software.

A Screenshot of the successful deployment on device is provided in the `screenshots` folder.