Writeup for cross Development electron project

Introduction

For electron, I converted my ionic project into a electron project, this way I could reuse most of my code from the ionic project (except the alerts, these conflict with electron's async architecture). The electron app just loads the ionic project inside the native windows chromium renderer. because of the nodejs api call and IPC requirement, I made use of IPC to save the movies to a json file in the localappdata folder and in order to access the filesystem, I used the nodejs fs module.

Project requirements

The app had to meet the following requirements:

- IPC between main and render process (tab 1 calls preload to save/load movies which then calls main process through IPC)
- Do (useful) Node.js api call (inside index.js)
- Security Electron
- Create an installer

I made use of IPC to save and load the movies to and from a json file in the localappdata folder (movies.json), in order to access the filesystem I used the nodejs fs module. The electron security points are listed down below. And lastly I created an installer using squirrel.

All requirements were met successfully.

Setting up the project

To set up the development environment I first installed electron globally using `npm install -g electron`. Then I copied my ionic project into a new folder. After this I ran `npm install` in the project folder to install all dependencies (these were removed for uploading my ionic project on canvas).

From this point on I followed the steps in the tutorial markdown file to set up electron.

I created the ElectronMain folder and did `npm init` to create a package.json file, for the questions I just pressed enter to accept the defaults. I checked if the results in the markdown file and my package.json were the same, which was the case, and went on with creating the index.js and preload.js files and copied the code from the tutorial markdown files.

Lastly I changed the base href in index.html from `<base href="/">` to `<base href="./">` so that the paths would be correct when loading the ionic project in electron.

The app can now be started by first running `ionic cap copy` to build the ionic project and copy the files to the www folder, and then running `electron electronmain` to start the electron app.

The `ionic cap copy` command gave the following error: `npm.cmd i -D -E @capacitor/cli@latest exited with exit code 1.`, this is because of version mismatches with capacitor, but since capacitor is not needed for this project I just ignored the error and let it be.

I also created an interface for easier access to the api, the content of this interface was also in the tutorial markdown file. I only had to add my own methods for saving and loading movies (and remove unnecessary ones).

Development of the app

For electron not much development was needed since I could reuse most of my ionic code. The only things I had to change were the alerts to toast messages since alerts dont work in electron, they would freeze up the app when they popped up and after closing the app would still be frozen. I also implemented the ability to save and load movies using IPC and the nodejs fs module.

Creating the installer

In order to create the installer I used electron-forge with the squirrel maker. In order for electron-forge to work I removed the package.json and node_modules folder from the ElectronMain folder. Then I checked if the package.json file in the root of the project had the correct main entry point and if it contained all necessary data. I also had to change the index.js loadFile path to point to the correct location of the built ionic project (./www/index.html).

Then I finally installed electron-forge using:

```
npm install --save-dev @electron-forge/cli
```

I tried initializing electron-forge using:

```
npx electron-forge import
```

But this gave errors because of a version conflict with capacitor, earlier is decided not to change the version but now I was forced to do so. Once changed I ran `npm install` again to install the correct capacitor version. And initialized electron-forge again, this time it worked.

To create the actual installer I ran:

```
npm run make
```

The app gets installed correctly and runs without issues. The installer can be found in the out/make/squirrel.windows/x64/ folder. When installing it opens a blank window for a split second and then opens the app. The movies are saved inside the localappdata folder (movies.json).

Link to the theory

As mentioned in the theory, an electron app has 2 processes, the main process and the render process. The main process, index.js in my project, creates the desktop window and loads the build of the ionic app

into it. Then the preload script gets executed, this is an isolated bridge between the renderer en the desktop window, it only exposes the mentioned API's so the renderer only gets what it strictly needs. The render process then runs the ionic app inside this desktop window.

In my application, IPC is used to tell the main process when it needs to save or load the movies

Electron Security 20 points:

1. Only load secure content: Is implemented. I only load local content from the built ionic project, no remote URL's, which is secure because I wrote it myself.
2. Do not enable Node.js integration for remote content: Is implemented. nodeIntegration is not enabled (false by default), I have contextIsolation enabled, so the ionic app cant even access nodejs or elecrtron. If I were to load remote content it would still not have access since the renderer cant access nodejs or electron because of the contextIsolation.
3. Enable context isolation in all renderers: Is implemented. Inside the index.js file contextIsolation is set to true. This ensures that the preload script will run in a sperate context from the renderer. This forces us to expose API's to the renderer explicitly, which is more secure.
4. Enable process sandboxing: Is not implemented. Since both points 2 and 3 are already implemented, the app is already secure enough. Enabling sandboxing would add OS-level security features, but would also add complexity to the app.
5. Use ses.setPermissionRequestHandler() in all sessions that load remote content: Cannot be implemented. I load no remote content, only local files, so I cannot implement this.
6. Do not disable webSecurity: Is implemented. webSecurity is set to true by default, since I didn't change it, it's enabled.
7. Define a Content-Security-Policy and use restrictive rules (i.e. script-src 'self'): Is not implemented. But because I only work with local content and that angular already escapes dangerous content, the app is still secure.
8. Do not enable allowRunningInsecureContent: Is implemented. This is enabled by default, it prevents the loading of HTTP content inside HTTPS pages. But I only load local content so this setting is irrelevant.
9. Do not enable experimental features: Is implemented. I have not enabled any experimental features in my app so they cannot produce any bugs or unexpected behavior.
10. Do not use enableBlinkFeatures: Is implemented. Same as point 9, I have not enabled any experimental features in my app.
11. `<webview>`: Do not use allowpopups: Cannot be implemented. I do not use any webviews in my app, so this point is irrelevant.
12. `<webview>`: Verify options and params: Cannot be implemented. Same as for point 11, I do not use any webviews in my app.
13. Disable or limit navigation: Is implemented. I added code to index.js to prevent navigation in general since angular handles the routing in the app, this way the app cannot be tricked into loading remote content.
14. Disable or limit creation of new windows: Is implemented. It blocks all attempts to open new windows. These windows could lead to remote content, so blocking them increases security.
15. Do not use shell.openExternal with untrusted content: Cannot be implemented. I dont use shell.openExternal in my app, so this point is irrelevant.

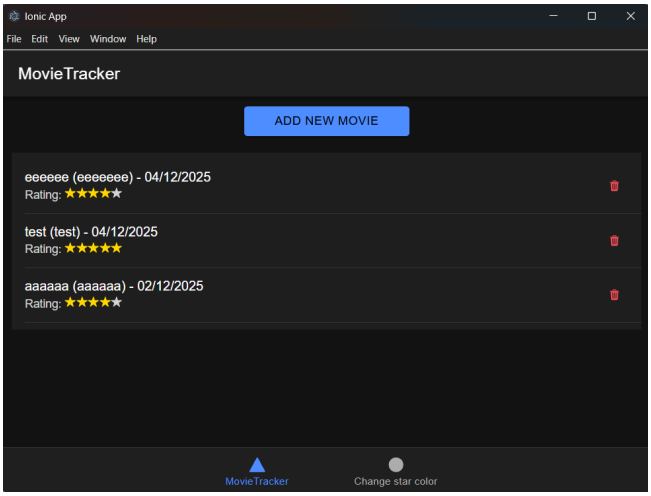
16. Use a current version of Electron: Is implemented. I use the latest stable version of electron which is 39.2.6 (package.json).
17. Validate the sender of all IPC messages: Is not implemented. Since my app uses local files only the risk is low to non-existent. No untrusted senders can send IPC messages to the main process.
18. Avoid usage of the file:// protocol and prefer usage of custom protocols: Is not implemented. In index.js I load the ionic project using LoadFile, which uses the file:// protocol. But since I already implement the navigation blocker all navigations will be blocked
19. Check which fuses you can change: Is not implemented. Because my app is just for learning purposes and not for production and will not be distributed, I did not create any fuses. But if I were to distribute the app I would create fuses to disable/enable features like RunAsNode, EnableCookieEncryption.
20. Do not expose Electron APIs to untrusted web content: Is implemented. My app doesnt expose any electron or nodejs API's. I only whitelist specific API's using the contextBridge in preload.js.

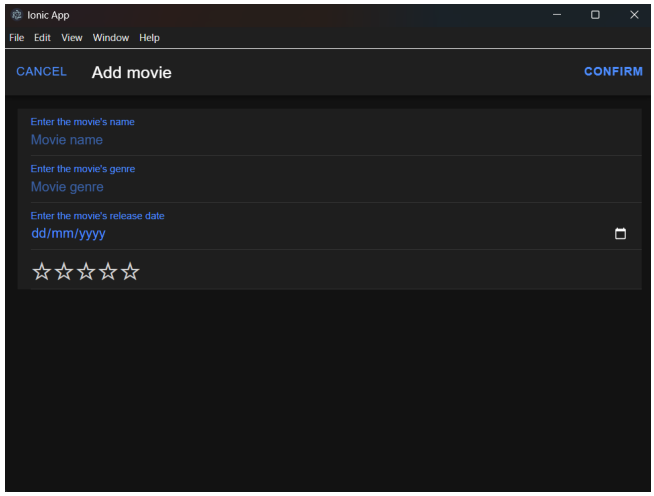
changes to the concept of the app as a result of the used technology

No changes were made to the concept of the app, I just adjusted/added some features to fit the requirements of the project. Like loading and saving movies using IPC and the nodejs fs module and also replacing the alerts with toast messages.

The end result

My app became a fully functional movie rater/tracker that runs an ionic project inside an electron app. The app allows users to add movies with a title, description and rating, view the list of added movies, and delete movies from the list. The movies are saved to a json file in the localappdata folder using IPC.





Conclusion

Electron is a great framework for building cross-platform desktop applications using web technologies. It allows developers to use their web development skills to create native desktop apps that can run on Windows, macOS, and Linux. The development process was straightforward, with all given documentation getting the ionic app to run inside electron was very simple. Overall, I am satisfied with how electron allowed me to repurpose my existing ionic app and create a desktop application with minimal effort.

Comparison with project in Qt and Ionic

Compared to the Qt and ionic versions of the app, the electron version has some advantages and disadvantages.

Compared to the Qt version, the electron version same as ionic is much easier to develop because it uses web technologies that I am already familiar with. The electron version also has better cross-platform compatibility, as it can run on Windows, macOS, and Linux without any additional effort. however the electron version will be slower since the Qt version is a native application and electron is a hybrid one.

Compared to the ionic version, the electron version provides a native desktop experience, which is more suitable for desktop users. The electron version can also access native desktop features and APIs that are not available in the ionic version. However, the electron version will require more resources than the ionic version, because ionic is optimized for mobile devices while electron is designed for desktop applications. Also, for the electron version, users will have to install the app while the ionic version can just run inside a web browser.