



Controlling a Browser for Google Login in an Obsidian Plugin

Background: You want an Obsidian plugin to obtain a Google credential by having the user log into Google, and you need *full programmatic access* to the resulting browser cookies (especially an OAuth session cookie). However, doing this inside Obsidian is challenging because the default embedded browser (iframes) has cookie restrictions. Below we explore all possible methods – both standard (orthodox) and unconventional – to control a browser for Google sign-in and retrieve cookies, on macOS (first) and Windows (similar principles), with Linux as a bonus.

Challenges with Embedded WebViews in Obsidian

By default, Obsidian's built-in *iframe*-based web viewer **disables cookies** for security ¹. This means simply embedding a Google login page in an Obsidian note via `<iframe>` will fail – Google will complain that cookies are turned off (preventing login). In fact, users have observed “Your browser has cookies disabled” errors when trying to log in via Obsidian iframes ². Obsidian uses its own Chromium instance and **does not share cookies with your external browser**, and it sandboxes iframes with no cookies by default ¹.

- *Solution:* Use a **custom embedded browser** approach that allows cookies. The community **Custom Frames** plugin (and Obsidian's core **Web Viewer** in newer versions) works around this by not using a vanilla iframe. Instead it creates a browser pane with cookies enabled ³ ⁴. According to Obsidian's documentation, when using the Web Viewer, third-party plugins *can* access its cookies: “While Obsidian is running, third-party plugins have full access to cookies in Web viewer” ⁵. This means your plugin can retrieve cookies from pages opened in that viewer.

However, **Google's OAuth login flow may not cooperate** with embedded webviews. Google announced in 2021 that it will *block* OAuth requests in embedded browsers (webviews) for security reasons ⁶. In other words, Google tries to prevent apps from using in-app webviews to sign in, because a malicious app could intercept the login info or session cookies ⁷. This policy began rolling out for OAuth clients in 2021 and can cause Google to refuse the login if it detects an embedded context. (The user might see a message like “You can't sign in from this screen” on an embedded login.)

Bottom line: Simply embedding Google's login inside Obsidian is problematic – you'd need a non-iframe browser view with cookies enabled, and even then Google might block it. Below are various approaches to tackle this:

Overview of Approaches

1. **Use Obsidian's Web Viewer/Custom Frame:** Open Google's login in an Obsidian pane (not an iframe) that allows cookies, then read the cookies via the plugin.
2. **External Browser + OAuth Redirect:** Launch the user's default web browser for Google's OAuth, then catch the auth token or cookie via a redirect (e.g. to a localhost server or custom URL scheme).

3. **Automation with Puppeteer/Playwright:** Programmatically launch a controlled browser instance (Chromium/Chrome) using a library, let the user (or script) log in, and extract cookies via code.
4. **Leveraging Electron BrowserWindow:** Since Obsidian is an Electron app, use Electron's ability to spawn a BrowserWindow or webview tag with a custom session, and then use Electron's session API to get cookies.
5. **OS-Level Automation:** Use system-specific automation to drive a browser: e.g. AppleScript on macOS to control Safari/Chrome, or WebView2/COM automation on Windows to automate Edge/IE. Retrieve cookies through those mechanisms.
6. **Direct Cookie Store Access (Unorthodox):** With user permission, read cookies directly from an installed browser's profile on disk (bypassing a live browser). This is **not** recommended but technically possible in some cases [8](#) [9](#).

We'll now dive into each method, with pros, cons, and implementation notes.

1. Embedded Browser in Obsidian with Cookies Enabled

Description: Use Obsidian's own UI to display the Google login page in a pane, but ensure cookies are allowed and accessible. The Custom Frames plugin is one example of creating an embedded browser pane (using Electron's `<webview>` under the hood) that supports cookies [10](#). Your plugin could similarly create a webview or utilize the core **Web Viewer**.

How to implement: In an Obsidian plugin, you likely have access to the DOM and possibly Node/Electron API. You could open a **WebView element** or a popup window for the Google login URL. Make sure to use a *partition* or session that allows cookies (the Custom Frames plugin does this, avoiding the normal sandboxed iframe). After the user logs in, you can retrieve cookies by either:

- Injecting JavaScript into that webview to read `document.cookie` for relevant Google domains, or
- Using Electron's session cookies API to query cookies for the webview's session.

Obsidian plugins (which run with full privileges) can get to the cookies. For example, if you can get the Electron `session` object for the webview, you can call `session.cookies.get({ url: "https://accounts.google.com" })` to get cookies [11](#). The Obsidian help explicitly states plugins have full cookie access in the Web Viewer [5](#).

Caveat – Google OAuth in WebView: As mentioned, Google may actively block the OAuth page in an embedded context [6](#). This means even if cookies work, the Google sign-in page might not allow the login to proceed. You might encounter an error or a blank page once credentials are entered. If that happens, you'll need to fall back to using an external browser (next approach) because Google trusts the system browser more than any embedded one.

In summary, *if* it works, this method keeps everything within Obsidian and gives you direct cookie access. It's worth trying on macOS (Obsidian's webview uses Chromium, so it's a modern engine) – just ensure you use a proper webview that isn't sandboxed. On Windows, the behavior is similar since Obsidian is the same Electron app.

References: Obsidian's iframe cookie restriction and Custom Frames workaround [2](#) [4](#); plugin cookie access [5](#).

2. Using an External Browser with OAuth Redirect (Standard Method)

Description: This is the orthodox OAuth approach. Instead of embedding at all, you open the user's **default web browser** to Google's OAuth URL, then get the result back via a redirect URI. Essentially:

- Your plugin opens a URL like `https://accounts.google.com/o/oauth2/v2/auth?...redirect_uri=http://localhost:PORT/...` in the default browser (maybe using Obsidian's `open link with default browser` capability or a Node call to `shell.openExternal`).
- The user signs in on their regular browser, which Google will allow (since it's not an embedded webview).
- Google then redirects to the `redirect_uri` you provided (which could be a localhost server your plugin is running, or a custom URI scheme that your app handles). This redirect contains an OAuth **code** or token.
- Your plugin catches this redirect (for example, by running a tiny HTTP server on `localhost:PORT` listening for the callback, or registering a custom protocol like `obsidian://auth-callback` and handling it).

At the end, you have an OAuth authorization **code or token** which you can exchange for credentials (access token, refresh token, etc.). You *don't directly get a cookie* this way, but you get the intended credential in a safer manner.

Pros: This method is Google's **recommended** flow for desktop apps (OAuth for installed apps). It leverages the user's actual browser – so all their password managers, 2FA devices, etc., work seamlessly ¹², and it complies with Google's security policies. You avoid dealing with low-level cookies entirely (Google gives you an OAuth access token or ID token instead).

Cons: If your use-case truly needs a **cookie** (for example, to call Google web endpoints that expect a logged-in session cookie rather than using OAuth tokens), this approach doesn't hand you the cookies from the browser. You'd have to replicate that session or use the token in authorized API calls. In many cases, though, an OAuth access token can be used to call Google's APIs instead of relying on a cookie.

Implementation notes: On macOS, you can use a custom URL scheme (via Electron's `app.setAsDefaultProtocolClient`) to catch the callback ¹³ ¹⁴. On Windows, similarly register a protocol or just use `http://127.0.0.1` with a known port for the redirect. The plugin can spawn a local server (Node's `http` module) to listen for the incoming GET request with the code. Once you grab the `code` from the URL, you exchange it with Google's OAuth endpoint for tokens – no need for cookies at all.

Conclusion: This is the most *orthodox* solution and most secure. If possible, use this method – it avoids the need to "get all cookies" manually. (Google's own documentation encourages OAuth flows for desktop apps rather than sharing browser cookies ¹⁵.)

3. Automating a Browser via Puppeteer or Playwright

Description: A more unorthodox but effective approach is to use a **headless/automated browser library** like Puppeteer (for Chromium/Chrome) or Microsoft Playwright. These allow your plugin to **launch a browser programmatically**, control it (navigate, fill forms, click buttons), and then inspect things like cookies from within your code. Importantly, you can run the browser in **headful mode** (not truly headless) so the user can see and manually complete the login if needed, but you still retain control to extract cookies.

How it works: From the Obsidian plugin (which can likely import Node libraries), you would do something like:

- `const browser = await puppeteer.launch({ headless: false });` - launches a Chromium browser. On macOS/Windows this will open a new browser window (separate from the user's main browser) under the control of Puppeteer.
- Open the Google auth URL: `const page = await browser.newPage(); await page.goto("https://accounts.google.com/...");`
- Either let the **user** input their credentials in this controlled browser window, or automate it (Puppeteer can `.type()` into fields, etc., but automating Google login may trigger bot detection or CAPTCHA - so it's usually better to let the user handle the actual typing with their 2FA, etc.).
- After the login completes, Google will redirect to the target service or show some page (for example, maybe you navigated to a Google consent page or something). At that point, the browser will have all the Google cookies (like `SSID`, `SID`, `OAuth` tokens in cookies, etc.) stored in its session.

Now you can **extract cookies** using Puppeteer's API. Puppeteer provides `page.cookies()` for the current page's cookies, or even `page._client.send("Network.getAllCookies")` to get all cookies in the session (including those from other domains) ¹⁶. For example, one script uses:

```
let currentCookies = await page._client.send("Network.getAllCookies");
fs.writeFileSync("cookies.json", JSON.stringify(currentCookies));
```

This dumps all cookies after the login (as shown in a Puppeteer login example) ¹⁶. You could then filter for the Google auth cookie you need (or just use the relevant cookie directly from memory).

Playwright offers a similar capability: after login, `context.cookies()` will give all cookies for that browser context ¹⁷. Both libraries allow saving and loading cookies for session persistence, but in your case you might not need to persist - you just want the OAuth cookie immediately and can then dispose of the browser.

Pros: This gives **full control** and full visibility. You essentially have a mini-browser you own: you can see all network traffic, cookies, etc. It's cross-platform (works on Mac, Windows, Linux in the same way). Also, by using a real Chromium engine, Google will see it as a *real Chrome browser* (especially if you set `headless: false` and maybe adjust the user-agent to Chrome's). That means it likely bypasses the "blocked webview" issue - from Google's perspective, it's just another Chrome instance. (In fact, it *is* - Puppeteer typically downloads a compatible Chromium build to use.)

Cons: The downside is complexity and overhead. Puppeteer adds a large dependency (Chromium ~100MB). Controlling the browser via code is more complex than just opening a URL. Also, if the user needs to interact, you have to communicate to them to use the Puppeteer-launched window. (The window might not have the usual browser UI or profile – it's a fresh session.)

Another consideration: Google's sign-in might challenge automated browsers (CAPTCHAs) if it suspects automation. To reduce that risk, run in non-headless mode and do not set the `--headless` flag; also possibly use `executablePath` to point to the user's installed Chrome and use a fresh profile directory so it looks like a normal browser. This way, the user can manually solve any 2FA/CAPTCHA. Essentially you're using Puppeteer as a controlled browser *framework* rather than a fully headless bot.

Example usage: A Medium article demonstrated using Puppeteer to log in via Google OAuth and scrape cookies ¹⁸ ¹⁹. After filling the login form (with email/password), they called `Network.getAllCookies` and saved the cookie data to a file ¹⁶ – confirming that you can indeed capture the Google session cookies this way.

Playwright vs Puppeteer: Both are viable. Playwright has the advantage of being able to automate *real* browsers (not just its bundled Chromium) – e.g. you could launch the user's installed Chrome or Edge with a user profile. However, using the user's actual profile is not recommended (you don't want to mess their real login state), but you can use a **persistent context** to keep cookies in memory. On Windows, Playwright can even automate Edge or Firefox if needed. Puppeteer is Chrome/Chromium-focused.

Wrap-up: This method is somewhat “headless automation” oriented, but with the user's permission it's a powerful way to get the cookies. Once you have the needed cookie (say an OAuth session token cookie), you can shut down the browser (`browser.close()`). Since you mentioned you don't need to persist the session, that's fine – just grab the cookie value and use it for whatever request or API call you need, then discard.

4. Using Electron's BrowserWindow and Session APIs

Description: Obsidian itself runs on Electron, which means you have access to the underlying Chromium engine. Rather than pulling in Puppeteer, you might be able to leverage **Electron's own browser control**. This approach lies somewhere between #1 and #3 in spirit: you create a new `BrowserWindow` (or use an offscreen `BrowserView`) from the plugin, navigate it to Google login, and then use the Electron session to get cookies.

Feasibility: Obsidian's plugin API isn't officially documented for creating new Electron windows, but since plugins are not sandboxed and have “deep control” ⁵, you can likely `require('electron')` in the plugin script. If that works, you can do something like:

```
const { BrowserWindow, session } = require('electron');
let authWin = new BrowserWindow({ width: 500, height: 600, webPreferences: {
partition: 'persist:googleAuth' } });
authWin.loadURL("https://accounts.google.com/o/oauth2/v2/auth?...");
```

Using a custom `partition` (or leaving it blank for a non-persistent session) ensures a fresh profile for this window. The user will log in on this window (it's a standalone window outside of the normal Obsidian panes – though you could style it as a child window).

After login, you can detect the success by either:

- Watching for a specific redirect URL (e.g., if using OAuth, Google will redirect to some success page or your custom scheme - you can listen to `authWin.webContents.on('will-redirect', ...)` or `did-finish-load` events to know when the login is done).
- Once done, use `session.defaultSession` or the specific session for that window to retrieve `cookies`. For example: `session.fromPartition('persist:googleAuth').cookies.get({})` returns all cookies in that session²⁰. You could filter by `domain: ".google.com"` or similar to just get Google cookies.

Electron's Cookies API allows querying by URL, domain, name, etc., and returns an array of cookie objects (with name, value, domain, etc.)²¹²². This way, you get the full cookie details programmatically.

Pros: No external dependencies – using Electron which is already running. It's also fairly direct: you control the window and can even customize it (e.g., disable popups, etc.). On Windows, this is essentially equivalent to using a Chromium WebView (since Electron uses Chromium under the hood). On macOS, same thing (since Electron is cross-platform).

Cons: Depending on Obsidian's environment, you might face some hurdles:

- Obsidian might not want plugins spawning arbitrary windows (for UX reasons). Ensure the user understands a new window will open. Possibly use `BrowserWindow({parent: remote.getCurrentWindow()})` to tie it to Obsidian's window.
- Google's embedded webview blocking `could` apply here as well. Electron's user agent by default might be recognized as "Chrome/<version> Electron/<version>". Google `might` detect the `Electron` part. A trick is to set the user agent to a vanilla Chrome UA for that window (`authWin.webContents.setUserAgent(...)`) to avoid any red flags.
- Security: Once the user logs in, *your plugin* has access to all those cookies (which is what you want, but also what Google is wary of). Make sure to handle and store that cookie securely or immediately use it then clear it.

Overall, this method is quite similar to approach #1 but implemented manually by your plugin rather than using Obsidian's built-in Web Viewer. It's essentially creating an **Electron mini-browser**. Many Electron apps implement OAuth this way: open a window for login and listen for the redirect to a known URL (some even use `electron.remote.dialog` if needed to show popups, etc.).

Given that Obsidian plugins have "deep control", this is a valid approach.

5. OS-Level Browser Automation (AppleScript, WebView2, etc.)

If you prefer to leverage the **user's actual installed browsers** or OS-provided web controls, there are platform-specific methods:

On macOS: You can use **AppleScript** (or JXA – JavaScript for Automation) to control Safari or Google Chrome with the user's permission. For example, you could write an AppleScript that tells Chrome to open a new window to the Google login URL. After the user signs in, you could attempt to retrieve the cookies. However, retrieving cookies via AppleScript is non-trivial: Safari/Chrome AppleScript APIs don't directly expose

cookies. You might try injecting JavaScript to get `document.cookie` from the page. Chrome does allow running JavaScript via AppleScript *if you enable a special setting*: the user must go to **Develop → “Allow JavaScript from Apple Events”** in Chrome’s menu ²³. Without that, AppleScript can’t run JS in Chrome for security reasons. Safari has a similar restriction (it generally disallows automation from reading page content unless you enable automation for Safari).

So, in practice, an AppleScript approach might look like: - Tell Chrome to open URL and activate. - Possibly display a dialog in your plugin asking the user to complete login and then press a button. - Once they press it, your plugin runs another AppleScript to execute `javascript "document.cookie"` in that Chrome tab (assuming JavaScript from Apple Events was enabled). This could return the cookie string for that domain, which you then parse.

This is quite **unorthodox** and fragile. It requires user to enable a setting in Chrome (or you fall back to UI scripting, which means simulating keystrokes to copy the cookie via dev tools – not ideal). There is also the requirement that the Obsidian plugin would have to run the AppleScript (this can be done via Node’s `child_process.exec("osascript ...")`). The user will likely have to grant Obsidian automation privileges in System Preferences if you start scripting other apps.

On Windows: A roughly equivalent approach is using the **WebBrowser control or WebView2**. Windows has a COM automation interface for Internet Explorer (through the old WebBrowser ActiveX). That could be used to navigate to Google and read cookies via the WinINet API (e.g., `InternetGetCookieEx`) or COM methods. However, IE is outdated and may not even allow Google’s modern login (it might break or Google might block it due to security requirements).

A better modern approach is **Microsoft Edge WebView2**, which is a Chromium-based control you can embed in any application. There are APIs to get cookies from a WebView2 instance: for example, WebView2’s `CoreWebView2.CookieManager.GetCookiesAsync` can retrieve all cookies after navigation ²⁴. If your plugin can interface with native code or if there’s a Node module for WebView2, you could launch an off-screen Edge WebView, display the login (even in a temporary window), then call `GetCookiesAsync` to fetch cookies. This is essentially similar to using Electron, but leveraging Microsoft’s library on Windows.

Another automation route on Windows is using **Selenium WebDriver** with a real browser. For instance, you could automate an existing installation of Chrome, Edge, or Firefox by driving it with Selenium (or a headless driver). Selenium’s WebDriver API has methods like `driver.manage().getCookies()` to pull cookies from the browser session once logged in. However, bundling Selenium (and a WebDriver binary) inside an Obsidian plugin is heavy and complex. If you were desperate to use the user’s full browser with extensions etc., you could instruct the user to install a specific extension or script to send the cookies to the plugin, but that’s beyond scope.

Pros of OS-level methods: In some cases, you’re leveraging the user’s normal environment (e.g., their default browser with possibly cached login). For example, a user might already be logged into Google in Safari or Chrome. Using AppleScript to navigate to a Google page might not even require credentials if their session is active; you could immediately get cookies. This could save the step of re-login. (Be mindful: Google often flags new login attempts from automation differently than existing session usage.)

Cons: These approaches are **fragile and platform-specific**. AppleScript is powerful but can break if any step fails or permission isn't granted. UI automation (like AutoHotkey on Windows or AppleScript UI scripting) can be considered invasive. Also, security software or OS protections might interfere (macOS might warn "Obsidian is trying to control Chrome" and require user consent).

In summary, OS-level automation is possible ("with user's permission" as you specified), but it's usually a last resort. It can achieve the goal (logging in a real browser and grabbing cookies), but with significant complexity in coding and user configuration.

Reference: AppleScript for Chrome requires enabling Apple Events JavaScript ²³. WebView2 cookie retrieval via CookieManager ²⁴.

6. Reading Browser Cookie Stores Directly (Very Unorthodox)

This method involves **no browser automation at all** – instead, with the user's permission, your plugin could try to directly read the cookies from the user's browser profile files on disk. For example, Google Chrome stores cookies in an SQLite database (`Cookies` file in the user's profile directory), but the cookie values are typically encrypted (on Windows, using DPAPI tied to the user account; on macOS, in the Keychain). Firefox also stores cookies in `cookies.sqlite` (unencrypted unless a master password is set).

In theory, one could locate these files, decrypt if possible, and extract the Google cookies. For instance, as one Stack Overflow answer noted, Firefox cookies (without a master password) are in a well-defined SQLite DB that you "could definitely try to read" ⁸. But the answer also immediately warns why this is problematic:

- **Security:** If any app could freely read another browser's cookies, it's a huge privacy hole. Anti-virus and security software often *prevent* or flag this behavior ⁹. Reading Chrome's cookie DB might require defeating encryption (which is possible if you call OS APIs from the user context, but not trivial).
- **Trust:** Users may consider it a breach of trust if your plugin siphons cookies from their browser profile without a clear understanding. It's akin to spyware behavior ²⁵.
- **Multiple browsers:** Which browser's cookies do you choose? The user might use Safari, Chrome, Firefox, etc. You'd have to handle each one's storage format and encryption. If multiple profiles or browsers have Google sessions, it gets messy ²⁶.
- **Upkeep:** Browser vendors change formats and encryption mechanisms over time, so this approach could break with updates.

In summary, this direct approach is *technically* a way to get a cookie, but it's **not recommended** unless you have a very controlled environment. It's better to have the user explicitly go through a login via one of the above methods. The Stack Overflow answer advising OAuth2 instead of cookie-stealing encapsulates the sentiment ¹⁵ – it's both safer and more user-friendly to use proper auth flows.

If you were to attempt it (for completeness): on Windows, one could use Windows Data Protection API (via a Node addon or PowerShell) to decrypt Chrome's `cookies` (the encryption key is tied to user login). On macOS, one could use the Keychain (via `security find-generic-password` commands) to decrypt Chrome cookies. This is complex and beyond typical plugin scope.

Conclusion

In conclusion, there are **multiple pathways** to achieve a Google login and retrieve the session cookies, each with trade-offs:

- **Preferred (Orthodox) Method:** Launch the system browser for OAuth and capture the token (no direct cookies needed). This aligns with Google's policies and avoids low-level cookie handling, but uses tokens instead of cookies.
- **In-App Browser Control:** Use Obsidian/Electron's embedded browser capabilities (Web Viewer or a custom BrowserWindow) to let the user log in within the app and grab the cookies via code. This gives full cookie access ⁵, but you must mitigate Google's blocking of embedded webviews ⁶ (e.g., by mimicking a real browser environment or falling back to external).
- **Automation Libraries:** Employ Puppeteer or Playwright to spin up a throwaway browser instance. The user can log in there (or automation can handle it), and you can programmatically extract all cookies once done ¹⁶. This works on macOS and Windows equally well, just at the cost of complexity and bundle size.
- **OS-Specific Hacks:** Use AppleScript on macOS or WebView2/COM on Windows to drive a browser or web control. This can work with the user's cooperation (e.g., enabling AppleScript JS injection ²³), but is less portable and more brittle.
- **Direct Cookie File Access:** Technically possible for some browsers ⁸, but strongly discouraged due to security and maintenance concerns ⁹.

Given that you only need a one-time OAuth cookie (no long-term session management), a safe bet is either **Method 2 (external OAuth)** or **Method 3 (puppeteer)**. These ensure you get the credential and stay within Google's supported methods. If you try the in-app route (Method 1 or 4), be prepared to handle potential Google sign-in restrictions.

By exploring all these approaches, you can choose the one that best fits your plugin's environment and the user's comfort level. Just remember to always obtain the user's consent, especially if employing any unconventional technique, since you are dealing with sensitive authentication data.

Sources:

- Obsidian iframe cookie restrictions and Custom Frames plugin ² ⁴.
- Obsidian Web Viewer plugin allowing full cookie access ⁵.
- Google's policy blocking OAuth in embedded webviews (for security) ⁶.
- Puppeteer automation for Google login and cookie extraction ¹⁶.
- Electron session cookies API (example usage of `session.cookies.get()`) ¹¹.
- AppleScript Chrome automation requires enabling Apple Events JS ²³.
- WebView2 cookie retrieval via GetCookiesAsync ²⁴.
- Stack Overflow discussion on reading browser cookies directly and its pitfalls ⁸ ⁹.

¹ ³ ¹⁰ Account log in through iframe display complain cookies disabled when it is not - Help - Obsidian Forum

<https://forum.obsidian.md/t/account-log-in-through-iframe-display-complain-cookies-disabled-when-it-is-not/38663>

2 4 [iframe - Enabling Cookies for Google Calendar Embedded Page in Obsidian Desktop App on Manjaro Linux - Stack Overflow](#)

<https://stackoverflow.com/questions/77297914/enabling-cookies-for-google-calendar-embedded-page-in-obsidian-desktop-app-on-ma>

5 [Web viewer - Obsidian Help](#)

<https://help.obsidian.md/plugins/web-viewer>

6 7 12 [Upcoming security changes to Google's OAuth 2.0 authorization endpoint in embedded webviews - Google Developers Blog](#)

<https://developers.googleblog.com/upcoming-security-changes-to-googles-oauth-20-authorization-endpoint-in-embedded-webviews/>

8 9 15 25 26 [javascript - How to get cookies from browsers to use in electron app? - Stack Overflow](#)

<https://stackoverflow.com/questions/75092726/how-to-get-cookies-from-browsers-to-use-in-electron-app>

11 20 21 22 [Class: Cookies | Electron](#)

<https://www.electronjs.org/docs/latest/api/cookies>

13 14 [How to implement google authentication in your Electron app? | by Arun Kumar | Medium](#)

<https://arunpasupathi.medium.com/how-to-implement-google-authentication-in-your-electron-app-aec168af7410>

16 18 19 [How to extract and login with Cookies from a site with Puppeteer ? | by Sujan Chhetri | Medium](#)

<https://medium.com/@sujanxchhetri/how-to-extract-and-login-with-cookies-from-a-site-with-puppeteer-23fbfc3b4ebf>

17 [Authentication - Playwright](#)

<https://playwright.dev/docs/auth>

23 [Applescript click automation in Google Chrome browser? - Ask Different](#)

<https://apple.stackexchange.com/questions/81062/applescript-click-automation-in-google-chrome-browser>

24 [Microsoft Edge WebView2 - Getting started | by Shyju Puzhakkal](#)

<https://medium.com/@shyjusachin/microsoft-edge-webview2-getting-started-13c9d40d8aa8>