ChatGPT

Integrating YouTube Music and Spotify Playback in Electron

Building an **Electron**-based desktop app with integrated music playback involves embedding the player for each service directly into your app's UI, using their provided web player APIs or SDKs. Below we outline integration approaches for **YouTube Music** (via YouTube's web player API) and **Spotify** (via Spotify's Web Playback SDK), along with pros/cons, code snippets (for renderer and preload contexts), and important licensing constraints. We also suggest fallback options (other services or local media) in case API limitations arise.

YouTube Music Integration (YouTube IFrame Player API)

YouTube does not offer a separate "YouTube Music" SDK, but you can use the **YouTube IFrame Player API** to embed YouTube videos or playlists (including music playlists) in your Electron app. This allows you to play music **within an Electron BrowserWindow** without opening an external browser. The IFrame API provides JavaScript control over playback – you can load playlists, play/pause, skip tracks, adjust volume, and even shuffle a playlist [1] [2]. Key steps and considerations:

- **Embedding a YouTube Playlist**: In your renderer process (e.g. an HTML/JS UI loaded in Electron), include the YouTube IFrame API script and create a player instance. For a playlist, use `playerVars` to specify the playlist ID. For example:

```
<!-- In your renderer HTML -->
<div id="ytPlayer"></div>
<script src="https://www.youtube.com/iframe_api"></script>
<script>
  let ytPlayer;
  window.onYouTubeIframeAPIReady = () => {
    ytPlayer = new YT.Player('ytPlayer', {
      height: '200', width: '300',  // YouTube requires at least 200x200 if
controls shown [3]
      playerVars: {
        listType: 'playlist',
        list: 'PLxxxxxxxxx',    // Your YouTube playlist ID
        autoplay: 0, controls: 0 // disable default controls (we'll control
via API)
      },
      events: {
        'onReady': () => { console.log('YT Player ready'); },
        'onStateChange': onYTStateChange
      }
    });
  };
  function onYTStateChange(event) {
    if (event.data === YT.PlayerState.ENDED) {
```

```
      console.log('Playlist finished');
    }
  }
</script>
```

In this setup, the player loads the given playlist (replace `'PLxxxxxxxxx'` with a YouTube playlist ID or a YouTube Music playlist's ID – YouTube Music playlists also have an ID usable in the YouTube embed URL). The `controls:0` param hides YouTube's UI; you can create your own play/pause buttons that call `ytPlayer.playVideo()` or `ytPlayer.pauseVideo()` as needed [1]. Likewise, use `ytPlayer.nextVideo()` and `ytPlayer.previousVideo()` to skip tracks [4]. The API also supports setting shuffle mode: call `ytPlayer.setShuffle(true)` before playing to randomize playback order [2] (and `setLoop(true)` if you want the playlist to loop continuously).

- **Playback Control via Preload**: If you have **context isolation** enabled in Electron, expose controls through the preload script. For example, in `preload.js`:

```
const { contextBridge, ipcRenderer } = require('electron');
contextBridge.exposeInMainWorld('musicAPI', {
  play: () => ipcRenderer.send('music-play'),    // main or renderer will
handle these
  pause: () => ipcRenderer.send('music-pause'),
  next: () => ipcRenderer.send('music-next'),
  prev: () => ipcRenderer.send('music-prev'),
  setVolume: (vol) => ipcRenderer.send('music-volume', vol)
});
```

In the renderer, you would attach these to UI controls, and in your main process or an IPC handler in the renderer, call the YouTube player's methods (`ytPlayer.playVideo()`, `ytPlayer.nextVideo()`, etc.). Alternatively, you can skip IPC and directly call the player from renderer if your design allows (e.g. by including the IFrame API in the same context). The **Electron BrowserWindow** should be configured to allow autoplay without user gesture. For example, when creating the window:

```
new BrowserWindow({
  webPreferences: {
    preload: path.join(__dirname, 'preload.js'),
    autoplayPolicy: 'no-user-gesture-required',  // allow autoplay
    backgroundThrottling: false                 // keep timers active when
hidden
  }
});
```

Setting `backgroundThrottling:false` ensures the music won't pause if the window is hidden or unfocused (the page stays "visible" even if minimized) – important for background playback [5]. You can also programmatically mute/unmute the **scenic video** content if needed. For example, if a YouTube video is playing the visual running track in another player or BrowserWindow, call its `player.mute()` method to silence it [6], or use `browserWindow.webContents.setAudioMuted(true)` to mute an

entire window. This way, your music playlist plays uninterrupted while the separate running video is silenced.

- **Pros of YouTube integration**: No login required for users (any public YouTube or YouTube Music playlist can be embedded), and the IFrame Player API is straightforward and powerful. You get full programmatic control over playback (play/pause, seek, volume, playlist traversal, etc.) in your app [1]. YouTube's library is enormous, so users can access virtually any song or playlist. Background playback is possible in Electron (unlike in mobile browsers, there's no built-in restriction on desktop aside from autoplay policy). You can also fetch playlist metadata (titles, thumbnails) via the YouTube Data API if needed for your UI – this requires a Google API key and possibly OAuth if accessing private playlists.

- **Cons and caveats**: YouTube will show **ads** in music videos for non-Premium users. These ads will play in the embedded player and you cannot programmatically skip or suppress them without violating terms. This could disrupt the music experience. Additionally, **YouTube's API Terms of Service** forbid separating the audio stream from the video [7]. This means you should not attempt to use YouTube just as an audio source by, for example, downloading or streaming only audio through unofficial means. Using the *official embedded player* (even if you hide the video element in the UI) is the compliant approach. Be aware that completely hiding the video (<iframe> set to zero size or CSS `display:none`) might violate YouTube's intent – at minimum, ensure the video is loading via the YouTube player and not a raw stream. Technically, the embed will function hidden (as developers often do for background music) [8], but from a policy standpoint it's safer to keep the video at least minimally present in the UI or clearly attributable to YouTube. Ensure you include YouTube's required attribution (the player itself will show the YouTube logo). Also note that YouTube content is **not guaranteed** to be available in all regions or forever – a video could be removed or region-blocked, causing playback errors for that track. Your app should handle errors (e.g., listen for the `onError` event from the player and maybe skip to the next video).

- **Performance**: Playing music via an embedded video is less efficient than a pure audio stream. The YouTube IFrame will load video data (though you can reduce quality by using the YouTube player API to set playback quality low if you want). However, for an MVP this is usually fine. If the Electron app has other heavy graphics (like a 3D running simulation), test that an extra embedded video doesn't cause frame drops. Running the player in a **webview or separate hidden BrowserWindow** is an option if you want to isolate it. For example, you could create a headless BrowserWindow that loads a minimal HTML with the YouTube player, and use IPC to send it commands (this is more complex, but keeps the main UI thread lighter).

- **Licensing and terms (YouTube)**: Using the official YouTube embed/player is generally allowed for personal or non-commercial app features, but if your app will be commercial, remember that the content is still subject to YouTube's licensing. You are effectively just streaming from YouTube on the user's behalf (which is allowed via the embed as long as you adhere to their API terms). Do **not** attempt to record or cache YouTube audio – that would violate content owners' rights and YouTube's terms. Also, if your app is distributed, you must follow the YouTube API Services terms. For example, you must not **modify or obscure** the YouTube player interface, and (as mentioned) you shouldn't provide a pure audio-only experience that removes YouTube branding or skips ads [7]. Since your use-case is an "immersive running app" (likely combining a video with music), be careful that you are not **re-packaging YouTube content** in a way that could be seen as circumventing YouTube. As long as users are logging in to YouTube (if needed) and viewing content through the official player, you should be fine. If you use the Data API to fetch

playlist info, note that Google's API terms require you to attribute and not store data longer than necessary, etc.

Spotify Integration (Spotify Web Playback SDK)

For Spotify, the best method is to use Spotify's **Web Playback SDK** – a JavaScript SDK that can create a Spotify player *inside* your application. This will stream music from Spotify's servers and allow full control (provided the user is a Premium subscriber). Unlike YouTube, Spotify's streams are DRM-protected and require authorization. **Key requirements** for Spotify integration:

- **Spotify Premium and OAuth**: Spotify **only** allows the Web Playback SDK for Premium accounts. The user will need to log in and authorize your app. You'll need to register a Spotify developer application (on the Spotify Dashboard) to get a Client ID and set up an OAuth redirect. In your Electron app, you'll go through an OAuth flow (for example, by opening a browser window to Spotify's authorization URL) to obtain an access token with the `streaming` **scope** (and likely `user-modify-playback-state`, and perhaps `playlist-read-private` if you want to list user's playlists). The Spotify docs emphasize that *"The Web Playback SDK needs an access token from your personal Spotify Premium account."* [9] So plan for a sign-in UI. Once you have a token, you can initialize the player.

- **Enabling Spotify playback in Electron (Widevine DRM)**: By default, Electron's Chromium **does not include Widevine CDM** (Content Decryption Module), which Spotify uses to stream encrypted audio [10]. This is a critical point – if you simply include the SDK script and try to play, it will fail to initialize or give a `playback_error` / `account_error` event. There are two main ways to solve this:

- **Use a Widevine-enabled Electron build**: A popular choice is the **castLabs Electron fork** (sometimes referred to as `electron-wvvmp` for Widevine). castLabs provides a custom Electron version with Widevine support built-in. You can install it via npm by specifying the castLabs repo and tag (for example, use `"electron": "github:castlabs/electron-releases#v25.3.1-wvvmp"` in your package.json to get a version matching a current Electron release, but with Widevine) [11]. This fork is maintained to track official Electron releases. Using it means your app will be able to play DRM content like Spotify or Netflix out-of-the-box.

- **Manually configure Widevine**: This approach involves extracting the Widevine CDM library from an installed Chrome browser and pointing Electron to it. Electron's documentation indicates you can specify `app.commandLine.appendSwitch('widevine-cdm-path', 'path/to/widevinecdm')` and the corresponding `app.commandLine.appendSwitch('widevine-cdm-version', 'x.x.x.x')` at startup. The `widevinecdm` binary and version must match what Chrome expects. This can be tricky and is not officially streamlined, especially across platforms. If you go this route, you'd need to ensure users have Chrome installed or package the CDM (which might have licensing issues). For an MVP, using castLabs' pre-packaged solution is simpler and legally safer (Google's Widevine CDM has terms of use, but castLabs likely handles the redistribution aspect for development).

- **Initializing the Spotify Player**: In your renderer code, after obtaining the OAuth token, load the Spotify Web Playback SDK. Spotify provides it via a URL; you can include: `<script src="https://sdk.scdn.co/spotify-player.js"></script>` in a renderer preload HTML or dynamically create a script tag. (Make sure your Content Security Policy or Electron `BrowserWindow` settings allow loading this script from Spotify's domain.) Once loaded, it will

provide a global `Spotify` object. Here's a **code snippet** for initializing and using the Spotify player in the renderer process (this could be in a script tag of your index.html or a bundled JS):

```
window.onSpotifyWebPlaybackSDKReady = () => {
  const token = '<user_access_token>';  // obtained from OAuth login
  const player = new Spotify.Player({
    name: 'My RunningApp Music Player',
    getOAuthToken: cb => { cb(token); }
    // volume: 0.5  (you can set initial volume here, 0.0 to 1.0)
  });

  // Error handling
  player.addListener('initialization_error', ({ message }) =>
console.error('Init Error:', message));
  player.addListener('authentication_error', ({ message }) =>
console.error('Auth Error:', message));
  player.addListener('account_error', ({ message }) =>
console.error('Account Error:', message));
  player.addListener('playback_error', ({ message }) =>
console.error('Playback Error:', message));

  // Playback status updates
  player.addListener('player_state_changed', state => {
    if (state) {
      const track = state.track_window.current_track;
      console.log(`Now playing: ${track.name} by $
{track.artists.map(a=>a.name).join(', ')}`);
    }
  });

  // Ready
  player.addListener('ready', ({ device_id }) => {
    console.log('Spotify SDK Ready with Device ID', device_id);
    // You can transfer playback to this device_id or start a playlist
    // e.g., use Spotify Web API to play something on this device:
    window.fetch(`https://api.spotify.com/v1/me/player/play?device_id=$
{device_id}`, {
      method: 'PUT',
      body: JSON.stringify({ context_uri:
'spotify:playlist:<playlist_id>' }),
      headers: { 'Content-Type': 'application/json', 'Authorization':
`Bearer ${token}` }
    });
  });

  player.addListener('not_ready', ({ device_id }) => {
    console.log('Device ID has gone offline', device_id);
  });

  player.connect();
```

```
  // Expose controls to window or via preload for UI buttons:
  window.spotifyPlayer = player;
  // (Then your UI buttons can call
window.spotifyPlayer.togglePlay(), .nextTrack(), etc.)
};
```

In this snippet: - We instantiate a new `Spotify.Player` with a device name and an OAuth token provider. The **access token** must be a valid token for a Premium user with the `streaming` scope [12] (Spotify's docs: *"You will need to execute this with a valid access_token string for a Spotify Premium user."*). - We attach event listeners for readiness and errors. Notably, `account_error` will fire if the user is not Premium or if there's an account issue (e.g., playback not allowed) – handle this by informing the user they need Premium [12] . - On the `ready` event, Spotify gives you a `device_id`. This represents the "virtual device" (Spotify Connect target) that your web player is running as. **Important:** *The Web Playback SDK cannot directly load a specific playlist or track without an external instruction.* The code above uses Spotify's Web API (the REST endpoint `/v1/me/player/play`) to start playing a playlist on the newly created device. We issue a fetch with the `device_id` and a `context_uri` for the playlist. This will start playback of that playlist on our SDK player. (Alternatively, you could load tracks via the SDK by URI if you have them, but using the Web API is straightforward for playlists or albums.) - After that, the user can control playback via the SDK's functions. For example, your app's "Pause" button could call `player.pause()` or `player.togglePlay()`, "Next" button calls `player.nextTrack()`, etc. The SDK supports shuffle and repeat *indirectly*: you set shuffle or repeat mode by calling the Web API (e.g., `PUT /v1/me/player/shuffle?state=true`). You might call those endpoints when the user toggles a shuffle UI button. The SDK's `player_state_changed` events will reflect the new state (the state object includes `shuffle` and `repeat_state` flags).

- **Integration with Electron Preload/Main**: Similar to YouTube, you can expose a control interface. However, since the Spotify SDK lives in the renderer (it's purely client JS), you can call it directly from renderer UI logic. For example, in your app's UI code:

  ```
  document.getElementById('playBtn').onclick = () =>
  window.spotifyPlayer.togglePlay();
  document.getElementById('nextBtn').onclick = () =>
  window.spotifyPlayer.nextTrack();
  // etc.
  ```

  If contextIsolation is enabled and you don't want to put the Spotify SDK in the preload, you might instead use contextBridge to expose limited controls that internally call `player.*` methods (similar to the `musicAPI` example earlier). Ensure that the Spotify SDK script is allowed to load; you might need to adjust the CSP meta tag or use `BrowserWindow`'s `webPreferences` like `sandbox: false` if issues arise loading an external script. In development, you can disable CSP for testing, but in production, whitelist the domains `https://sdk.scdn.co` and Spotify's token auth domain if used.

- **Pros of Spotify integration**:

- You're leveraging the user's Spotify **subscription** for high-quality, ad-free music. All playback is fully legal under Spotify's platform (the user must sign in and has rights to stream the music).

- **Programmatic control** is excellent: the Web SDK gives you events (track change, etc.) and controls (pause, resume, seek, volume, skip) [13] . You can sync UI elements (e.g., display the current song title/artist in your app) by reading the state events.
- You can access the user's own Spotify content: with additional Web API calls, you can retrieve the user's playlists, let them pick one from a dropdown in your app, etc. (This requires scopes like `playlist-read-private` or `playlist-read-collaborative` in the OAuth process if you want their personal playlists).

- No need to manage media content or storage – Spotify streams everything. Also, playback can continue in background (the Electron app can even be minimized to tray and music keeps playing, just like a native Spotify client, since our player runs in the background page).

- **Cons and challenges**:

- **Spotify Premium requirement** is a major limitation – users without Premium cannot stream on third-party devices. If your app targets a wide audience, this could exclude free-tier users (free users can use Spotify Connect only to control other devices, not to stream on new devices). The SDK will throw an `account_error` if a free account tries to connect [9] .
- **OAuth complexity**: You need to implement the OAuth flow in Electron. Typically, you'd open an external window (or use the built-in `shell.openExternal` to open the system browser) to Spotify's authorize URL, then have a redirect URI that your app can capture (e.g., using a custom protocol like `myapp://callback`). Electron's main process can catch this redirect and get the `code` to exchange for a token. This flow requires a client secret (if using Authorization Code flow) which you must keep safe (maybe do the token exchange on a backend server or use PKCE). Alternatively, you can use the Implicit Grant flow for a quick client-side token (less secure, but no server needed). For development, Spotify's dashboard lets you generate a temporary token manually to paste in (as shown in the quick-start). But in a real app, you'll automate the sign-in.
- **Widevine DRM**: As discussed, getting Widevine working can be a hurdle. The castLabs fork simplifies it but means depending on a forked Electron. Ensure the version you use is up-to-date with security fixes. In 2025, castLabs is still maintaining forks for major versions. If you need to support **Windows** users, note that some older widevine solutions weren't supported on Windows [14] – but castLabs **does** support Windows (as shown by the successful playback of protected content in the example) [15] [16] . Always test on all target OSes.
- **Playback control nuances**: The Web Playback SDK runs in the context of your app, but Spotify's service has some say in playback. For example, if the user is actively playing Spotify on their phone or another device, starting playback in your app might **take over** as the active device (or you might need to explicitly transfer playback). Similarly, when your app's player is active, it will show up in the user's Spotify account as a "device" called whatever name you gave (e.g., "My RunningApp Player"). This is usually fine (even cool to show the integration), but it's something to be aware of.
- **No offline mode**: You cannot cache or play music offline via the SDK. The user must have internet and the token must be refreshed at least hourly. Plan to handle token expiration (the `getOAuthToken` callback will be called by the SDK when it needs a fresh token – you should retrieve a new token using your refresh token in that case).
- **Licensing and restrictions (Spotify)**: **Spotify's developer terms explicitly forbid certain uses.** Notably, you *cannot charge* for a streaming service or redistribute music. In fact, Spotify's policy states *"Streaming applications may not be commercial"* [17] – you can't use the Spotify API to create a commercial music service. If your app is a **fitness app** that simply **allows the user to log in and play their own music**, this is generally allowed (you're not providing the music, the user's Spotify account is). But you should not, for example, bundle a "free music included" feature

without the user logging in to their own account. Also, Spotify forbids *synchronizing music with video or other media* [18] in a pre-arranged way. This is aimed at things like setting Spotify songs as soundtracks to videos or slideshows. In your case, if the user is just choosing a playlist to run to, and your visuals are not tied to specific songs' timing, you should be okay. Just don't do something like automatically switch scenes based on the song or create a video that's essentially a music video – that would violate the "no synchronization" rule. Another rule: you cannot **record or re-broadcast** Spotify streams (no saving the music, and no re-streaming it to others). All audio must go directly to the user's device for personal listening. The SDK enforces this (there's no way to grab raw audio data from it).

- **Platform considerations**: Because the SDK relies on WebAudio and EME (Encrypted Media Extensions), it might not run in all environments if the environment is constrained. In an Electron desktop app with Widevine, it should. But, for example, if you ever target the Microsoft Store, there could be issues (just something to keep in mind; as an MVP on regular desktop, it's fine).

- **Alternative Spotify integration (Connect API)**: If supporting only Premium users is too limiting, one partial workaround is to use Spotify's **Connect Web API** to control playback on the user's *Spotify app* (if they have it running on their phone or PC). With the Web API, you can list the user's devices and send play/pause/next commands to an official Spotify app. This would let a free user at least control an existing session (free accounts can play in shuffle mode on some devices). However, this approach does **not play music through your app** – the sound would come from the user's phone or Spotify client, which is likely not what you want in an "immersive app" (you'd want the music and visuals in one place). It's an option to mention, but probably not the desired UX. For an MVP, it's better to require Premium and note it in your app description.

Other Services and Fallback Options

If YouTube or Spotify integration proves problematic (due to API terms or user requirements), consider these alternatives:

- **SoundCloud**: SoundCloud offers a huge library of user-uploaded music, much of it free to listen. They provide an embeddable **widget player** for tracks and playlists, with a JavaScript Widget API for control. You can embed a SoundCloud playlist in an `<iframe>` and use `SC.Widget` to play/pause, skip tracks, adjust volume, etc [19]. For example, an iframe pointing to a playlist like `https://w.soundcloud.com/player/?url=https%3A//api.soundcloud.com/playlists/<playlist_id>` can be controlled via `SC.Widget(iframeElement)`. The Widget API supports methods like `widget.play()`, `widget.next()`, `widget.setVolume(vol)` and events for track end or play progress [19]. **Pros:** No login required for the end-user if the content is publicly available, and no premium subscription needed. SoundCloud has a lot of remixes, indie music, and some popular tracks (depending on licensing, many mainstream artists are not fully available, but you might find workout mixes, etc.). **Cons:** You are limited to content that's on SoundCloud (quality and availability vary). There might be API rate limits if you search or use their REST API (you'd need a client ID for API calls to get track info by name, etc.), though simple embedding might not require authentication. Also, SoundCloud's terms allow embedding their player, but if your app is commercial, you should review their developer terms – generally they are more permissive than Spotify for free content, but they wouldn't want you ripping audio either (the widget ensures streaming only). If your running app is just user-facing and not redistributing the tracks, SoundCloud embedding should be fine.

- **Local Media or Bundled Music**: A straightforward fallback is to allow users to play local audio files (MP3/AAC, etc.) through a built-in player in your app. Electron can easily handle local file playback with the `<audio>` or `<video>` HTML element or via Node modules. For instance, you could drop an `<audio>` element in your renderer and use the HTML5 Media API to control play/pause/seek. For playlists, you'd implement a file picker or let users drag-and-drop files/folders. **Pros:** No licensing issues if the user provides their own music (they are responsible for the content). This works offline and is simple. **Cons:** It doesn't integrate streaming services (so the user must have their own library of music files, which is less common nowadays). It also doesn't fulfill the desire to directly integrate YouTube/Spotify content, so it's more of a user-experience fallback.

- **Other streaming services**: There are a few other platforms:

- **Apple Music** has a Web Playback API (MusicKit JS) that allows web apps to play full Apple Music tracks if the user is an Apple Music subscriber. This is an option if your user base might have Apple Music. However, using it requires the developer (you) to obtain a MusicKit key from Apple and the user must sign in with their Apple ID. It's similar to Spotify in complexity (premium-only, DRM streams, etc.), and on Electron you'd face the same Widevine issue because Apple Music also uses DRM via FairPlay (which, on Windows, falls back to Widevine I believe for the web player). Apple's terms are also quite strict.
- **Deezer** offers a Javascript SDK that can stream music (Deezer has a "connect" similar service). They allow 30-second previews for free accounts and full streams for premium accounts. Deezer's SDK might not require heavy DRM (I'm not fully up-to-date, but they had an Flash-based widget long ago and now a newer API). If Deezer content is important and they have some open API, it could be a secondary option.
- **YouTube Music unofficial API**: Some developers use unofficial libraries like `ytmusicapi` (Python) or scrape techniques to get YouTube Music content programmatically. For example, using youtube-dl / yt-dlp to fetch an audio stream from a YouTube Music URL. This would let you play just audio through a normal <audio> tag. **This is not officially sanctioned by YouTube**. While technically feasible (yt-dlp can retrieve direct audio stream URLs for YouTube), it squarely violates YouTube's Terms (it **separates audio** from video and avoids the YouTube player/ads [7]). Apps that have tried this have been rejected from app stores or received cease-and-desist in some cases. So, it's mentioned here only as a last-resort hack for personal or closed-use testing, not for a released app.

- **Radio/Streaming**: There are free streaming radio APIs (e.g., Shoutcast streams or internet radio stations) that you could integrate for background music. These are just MP3/AAC streams you can play in an <audio> element. While not user-curated playlists, you could offer a few genre-based stations (there are directories of free streams). The benefit is simplicity (no login, just a URL to play) and usually no legal issue if the station is public. The downside is lack of user control (it's not on-demand playback) and you can't guarantee the music style matches what the user wants.

- **Caching and Performance Considerations**: Regardless of the service, you might consider caching certain data:

- *Playlist metadata*: It's a good idea to store the list of songs (titles, IDs) for a chosen playlist in local storage or a file once retrieved, so you can display it quickly next time. For example, if a user logs in with Spotify and selects a playlist, you could save that playlist ID and name, so next time the app loads you present it (and maybe auto-fetch an updated track list in background). For YouTube, if you use the Data API to get video titles in a playlist, cache them similarly. This avoids

re-calling APIs too frequently (and if offline, you at least know what songs were in the list, even if you can't stream them offline).

- *Preloading Next Track*: For gapless or smooth transitions, you might want to preload the next song slightly before the current one ends. With YouTube's iframe player, you don't have manual buffer control for the next video (YouTube handles the buffering internally – typically it won't preload the entire next video until the current is near end). There isn't an easy way to preload two YouTube videos simultaneously via one player. A hack could be creating a second hidden player and cueing the next video in it, but syncing them is complex and probably not worth it. Spotify's SDK doesn't give direct preload control either, but Spotify's backend is pretty fast at transitioning tracks, and they likely buffer ahead on their side when possible. So you might not need to do anything special; just be aware that first track start might have a small delay (especially if needing to transfer playback to the device).
- *CPU/Memory*: Embedding a YouTube player (Chromium <iframe>) or running Spotify's player will consume memory. Ensure you destroy or reuse players as needed. For example, if the user switches from a YouTube playlist to Spotify, you might want to `player.destroy()` the YouTube iframe to free resources. Or if you have multiple services loaded, be mindful of multiple audio outputs (make sure to pause one when starting another).
- *Audio focus*: Decide how to handle if the user tries to play music from both services at once or in conflict with the running video's audio. It's usually best to have a single audio stream at a time (aside from sound effects). So implement logic like: if user starts a Spotify playlist, pause/stop the YouTube playback (and vice versa). Also, if your running app itself has sound (e.g., ambient noises or coach voiceover), you might need to mix audio. Electron can manage multiple audio sources, but balancing volume or muting certain sounds when music is on might improve the experience.

Summary of Pros & Cons

To help you decide, here's a quick comparison of integration paths:

- **YouTube Music (via YouTube embed):** *Pros:* Free for users (ad-supported), no signup needed, enormous content library, straightforward JS API for control [1]. *Cons:* Ads may disrupt playback; requires internet (no offline); slightly hacky to use as "audio only" (must honor YouTube's terms); audio quality varies and there's no official guarantee of uptime or consistency for each video. Also, if your app is commercial, relying on YouTube content could be seen as piggybacking on YouTube's service (generally allowed, but you cannot charge specifically for that content). There is also a slight delay switching tracks compared to native music players, and potential for content to be removed by uploaders.

- **Spotify (Web Playback SDK):** *Pros:* High-quality, uninterrupted music tailored to user's taste (their playlists, etc.), rich API and ecosystem (you can leverage Spotify's metadata, album art, etc.), and it keeps your app in compliance with music licenses (Spotify handles royalties). *Cons:* Only for Premium users – big barrier to entry. More complex to implement (OAuth flow, token refresh, and Electron DRM setup). Also, **Spotify's API terms restrict commercial use** – if your app is a paid product, you should ensure you're not violating "no commercial streaming" rules [17]. Many hobby/fitness apps do use Spotify integration, but they typically don't monetize the music feature directly (it's seen as "connect your own Spotify"). You might want to add a disclaimer like "Requires Spotify Premium account" in your app description.

- **Other/Hybrid approaches:** *SoundCloud* can be a decent free alternative (no subscription needed), but content might not match user's desired mainstream tracks; also it can be integrated similarly with an iframe widget. *Local files* are a good backup to let users play

something if they have it, but expect that most users nowadays prefer streaming. *Radio streams* could provide hassle-free background music (with no login), but lack user control over song selection.

- **Legal/Permission summary**:

- **YouTube**: Permits embedding videos and playlists on third-party apps [20], but you must use their player. Do not attempt to bypass the player to grab raw audio (that's explicitly disallowed) [7]. If your app will be distributed, consider registering it for a YouTube API key (if you use Data API) and abide by the branding requirements. YouTube might require you to show a notice or have certain minimum functionality if using their API services – for instance, their API TOS says you should provide proper attribution and not obscure advertisements.
- **Spotify**: Requires end-user permission (OAuth) and only allows personal streaming. You, as the developer, must agree to Spotify's developer terms which include *not charging for streaming content, not modifying the music, and not synchronizing with video* [17] [18]. For a fitness app, these are usually fine if you just let users play their music. Just avoid hard-syncing music to your video content in a pre-authored way (e.g., don't create a feature where a certain playlist is tightly coupled with a specific video route such that it effectively becomes a soundtrack – unless you've ensured it doesn't violate the sync rule).
- **SoundCloud**: Their API terms allow streaming of tracks via their widget/player. If you use their API directly to get track URLs (MP3 streams), you must adhere to their guidelines (which typically say you shouldn't download or cache tracks without permission, and you must attribute SoundCloud). Using the official widget keeps you in the clear.
- **Others**: Each service has its own rules. Apple Music's terms, for example, would require your app to sign in an Apple user and you cannot use that content outside of Apple's authorized frameworks. In general, leveraging the **official web player or widget** for each service keeps you within permitted use.

By combining these integrations, your immersive running app can let the user choose a playlist from a dropdown in the UI (populate this by calling the respective APIs: e.g., get user's Spotify playlists via Web API, or allow them to paste a YouTube playlist URL, etc.), then play it seamlessly in the background. The user will be able to **control playback** with on-screen buttons you provide (mapped to the APIs or SDK calls), such as Play/Pause, Next, Previous, Shuffle, Volume and Mute.

Finally, ensure you test the experience thoroughly: for example, **mute the running video's audio** when music is playing (as you intended) – you can programmatically mute the video element or use separate audio contexts so they don't conflict. In Electron, multiple audio sources will mix by default, so you have to manually silence one. Using the techniques above (YouTube player's `mute()` for the scenic video player, or SoundCloud/HTML5 video element `.muted = true`) will handle that.

With these integrations, you should be able to achieve your MVP: an Electron app where the user can **select a music playlist in-app** (no external browser required), enjoy background playback of YouTube Music or Spotify tracks during their run, and have full control over playback (shuffle, skip, pause, volume) via your app's interface. Good luck with your build, and happy running!

Sources:

- YouTube IFrame Player API – official documentation on embedding and controlling YouTube players [1] [2] (supports loading playlists, controlling volume, shuffle, etc.).
- Excerpt from YouTube API Terms of Service prohibiting separation of audio/video [7] (relevant to not stripping audio from the video stream).

- Spotify Web Playback SDK – official docs on getting started (Premium account and token required) [9] and API reference (player control methods) [13].
- Spotify Developer Policy notes – no commercial streaming, no sync with visuals [17] [18].
- Medium article noting Electron's lack of Widevine DRM by default (requiring special Electron build for Spotify/Netflix) [10].
- castLabs Electron fork usage example (demonstrates enabling Widevine via custom Electron build) [11] [15].
- SoundCloud Widget API documentation (showing how an embedded SoundCloud player can be controlled via JS with play/pause/next, etc.) [19].

---

[1] [2] [3] [4] [6] YouTube Player API Reference for iframe Embeds | YouTube IFrame Player API | Google for Developers
https://developers.google.com/youtube/iframe_api_reference

[5] BrowserWindow | Electron
https://www.electronjs.org/docs/latest/api/browser-window

[7] copyright - Why an app that streams audio from third party site that extracts audio from Youtube is illegal? - Law Stack Exchange
https://law.stackexchange.com/questions/14854/why-an-app-that-streams-audio-from-third-party-site-that-extracts-audio-from-you

[8] javascript - How to add youtube music video playing in background in Html - Stack Overflow
https://stackoverflow.com/questions/44252762/how-to-add-youtube-music-video-playing-in-background-in-html

[9] [13] [17] [18] Getting Started with Web Playback SDK | Spotify for Developers
https://developer.spotify.com/documentation/web-playback-sdk/tutorials/getting-started

[10] [14] Spotify Web Playback SDK — feat. Electron | by Callback Insanity | Medium
https://alexanderallen.medium.com/spotify-web-playback-sdk-feat-electron-50ff930d5c74

[11] [15] [16] castLabs + Widevine CDM. Taking CastLabs Electron fork for a... | by Callback Insanity | Medium
https://alexanderallen.medium.com/castlabs-widevine-cdm-ea369bb5623b

[12] Web Playback SDK Reference | Spotify for Developers
https://developer.spotify.com/documentation/web-playback-sdk/reference

[19] Widget API - SoundCloud Developers
https://developers.soundcloud.com/docs/api/html5-widget

[20] Embed videos & playlists - YouTube Help
https://support.google.com/youtube/answer/171780?hl=en