ChatGPT

Introduction

Building a “virtual TV channel” app for LG webOS TVs (in the spirit of Plex-based apps like Coax and QuasiTV) requires carefully replicating the look-and-feel of linear cable TV using the user’s Plex media library. This report presents a comprehensive implementation plan and technical specification for such an app, focusing on scheduled playback channels with an Electronic Program Guide (EPG), robust Plex integration, and a performant webOS client. We draw on best practices from existing solutions (e.g. Coax’s zero-setup channel surfing and QuasiTV’s scheduling engine) and incorporate LG’s latest webOS development guidelines. The goal is a reliable, smooth user experience where a viewer can “channel surf” their Plex content as if it were live TV – complete with a channel guide, remote control navigation, and seamless playback continuity.

Key Objectives: The app will generate linear **scheduled channels** from Plex content and present them in an EPG UI. Users can flip channels with the remote, seeing whatever show or movie is “currently playing” on each channel. If a user joins a channel in the middle of a program, playback should start at that point (not from the beginning) to simulate true live TV. The app must handle **playback continuity** (automatically playing the next scheduled item when one ends) and provide standard DVR-like controls (pause, limited rewind, etc.) without breaking the linear illusion. Integration with **Plex** is critical: the app will authenticate the user’s Plex account, fetch library metadata (to know what to schedule and for how long), and retrieve stream URLs for playback (favoring direct play when possible). We also must optimize for **LG webOS TVs** (v4.0+), using appropriate technologies (HTML5/JS with LG’s Enact framework or similar), adhering to platform media capabilities, and meeting performance constraints (limited CPU/memory and older browser engines [1] [2]). Finally, we consider **deployment strategy** (hosted vs. packaged app) and compliance with LG’s app store policies, ensuring the app can be efficiently updated and approved for distribution.

Feature Set and User Experience

To match apps like Coax and QuasiTV, our virtual channel app will provide the following core features and UX elements:

- **Scheduled Linear Channels:** The app will automatically create a set of “virtual channels” from the user’s Plex library content. Each channel is essentially a playlist of shows or movies arranged in a continuous 24/7 schedule. For example, one channel might air episodes of a particular series or genre back-to-back, while another might mix various shows based on a theme or user-defined collection. The scheduling engine ensures each channel has a deterministic lineup at any given time – if it’s 8:45pm, a specific episode is playing on channel 5, just like real TV. This deterministic schedule enables a shared experience: “channels” aren’t just random shuffles; they behave like real broadcast channels with a known sequence of programs.

- **Electronic Program Guide (EPG):** Users can bring up a channel guide grid that looks and feels like a cable TV guide. This guide lists channels (with names/numbers) along the vertical axis and timeslots horizontally (e.g. 8:00pm, 8:30pm, 9:00pm, etc.), with each program represented as a rectangular

block spanning its scheduled duration. The guide is scrollable in time and channel list. QuasiTV's guide, for example, shows ~2 hours at a time and allows scrolling further out to see upcoming programs [3]. Our app will similarly support browsing the schedule. The currently airing program on each channel will be highlighted, and as time progresses the guide will update (e.g. auto-scroll or refresh to keep the current time indicator in view). Selecting a program (by pressing OK) will tune that channel immediately. Users can also navigate the guide with arrow keys (up/down to change channel row focus, left/right to move in time) and press a **"go live"** button to jump back to the present moment if they have scrolled away from now (a feature QuasiTV added in v2.7 [4] [5]).

- **Channel Surfing & Instant Tuning:** A critical part of the experience is fluid channel surfing using the remote. Pressing the channel **Up/Down** buttons on a conventional remote (or a corresponding UI control on the Magic Remote, since webOS may not deliver channel key codes to apps [6]) should flip to the next/previous channel. This should happen quickly, with minimal buffering, to mimic the rapid switching of cable TV. To achieve this, when the user changes channels, the app stops the current stream and starts the new channel's stream at the correct position. We may display a small overlay with the new channel's information (channel name/number, current show title, time) for a second, similar to a TV OSD, to confirm the change. Additionally, pressing an **"Info"** button or a specific remote key could bring up a smaller mini-guide or show info banner without leaving the current channel, allowing the user to see what's playing now and next on other channels.

- **Playback Start and Resume Behavior:** When tuning to a channel, playback should *start in progress* if the scheduled program isn't at the beginning. For instance, if Channel 3 is scheduled to play *Jurassic Park* from 8:00–10:00pm and you tune in at 9:00pm, the movie should start playing from the 1-hour mark (not from the beginning) [7]. This creates the authentic feeling that "the show was already playing" before you arrived. Under the hood, the app will calculate the offset into the media and seek to that timestamp after loading the stream. Conversely, if you stay on a channel across program boundaries, it should seamlessly transition to the next show according to schedule (with perhaps a brief on-screen "Coming up next: [Title]" message). If you leave a channel and return shortly after, the app should still honor the schedule (i.e. the program has continued to progress in your absence). There is **no concept of "pausing" the channel globally** – pause is user-specific. If a user pauses, the app will buffer content for them, but the channel's schedule keeps moving. When they unpause, they'll be behind live; they can catch up using "fast-forward" (if implemented as short skip increments) or a "live" button to jump to current playback. We will implement a **time-shift buffer** of limited length (perhaps ~30 minutes) to allow some rewind/fast-forward within the currently playing program. However, skipping ahead of the real-time position will not be allowed (the app can enforce that you cannot seek past the live point). This is similar to how live TV behaves on Plex's own DVR service or other IPTV solutions.

- **Channel Guide and Metadata Display:** For each program, the app will display metadata drawn from Plex – title, season/episode, episode synopsis, and if possible a thumbnail or background image. The EPG will show the program titles in each timeslot, and focusing on a program can show a description in an info panel. On the currently playing channel, an info bar can be invoked to show the title of the show/movie, an image (e.g. show poster or still frame), and its running time (e.g. *"S4E3 – 8:01 PM–8:22 PM"* as in QuasiTV's UI) [8]. We will also label channels meaningfully. QuasiTV initially auto-generated channel names from the TV show's original network (Studio) [9] – e.g. a "NBC" channel playing all NBC shows – and later allowed custom naming and grouping. Our app might allow channels based on **user-defined collections or playlists** (like QuasiTV 2.1 does [10]) or

automatically group content (e.g. a “Comedy Channel”, “Action Movies Channel”, etc.). Each channel can have a friendly name and number.

- **Remote Control Navigation:** The app is designed for remote input (“10-foot UI”). This means large, legible text and focusable UI elements. **Focus management** is crucial – as the user navigates the guide or menus with arrow keys, the currently focused item is clearly highlighted (e.g. a bright outline around a selected program in the EPG, or a distinct color for the active channel in a list). We will adhere to LG’s TV app design guidelines for focus visuals and not rely on hover or touch-style interactions. On LG’s Magic Remote, the pointer can be used, but we will ensure the app is fully operable with just the D-pad and OK/Back keys (since many will navigate that way) [11] [12]. The **color keys** (red/green/yellow/blue) on the remote can be optionally mapped to quick actions (e.g. a color key could toggle closed-captions, or open settings) [13] [14]. Standard **media keys** (Play/Pause, FF, REW) on conventional remotes will be supported as well – pressing Pause will pause the current channel’s playback, Play or OK will resume, and FF/REW keys could jump the playback a few seconds forward/back (within the buffered range) [15]. If the user long-presses the Back button, the app will handle it per webOS conventions (on newer webOS, long Back exits the app; short Back may open an in-app menu or go to a previous screen) [16].

- **Visual Design and Readability:** The UI will use large fonts and high-contrast colors suitable for a TV viewed from across a room. For instance, an EPG typically has a dark translucent background with light text. Focused items might be highlighted with a bright accent color or enlarged slightly. We will respect a **safe title area** (padding of ~20px inside the screen edges) so that no text or controls are cut off by TV overscan [17]. All text (show titles, times, etc.) should remain within this safe area. Additionally, we’ll ensure that if the user’s TV is set to overscan (rare on modern flat panels, but still), critical UI isn’t lost. The app will also include on-screen hints where appropriate (for example, showing an icon of the remote’s “OK” button in the UI next to “Watch” or a color button icon next to an action label, to teach first-time users how to trigger actions). We will incorporate branding and visual polish but avoid clutter – as LG’s design principles say, keep the interface clear and focused on content [18].

- **Multi-User and Profiles:** If multiple Plex users or profiles are relevant (for instance, a family Plex server with different users), the app could allow switching Plex accounts or servers. QuasiTV eventually introduced profiles to maintain separate channel lineups [19]. In our design, initially the app will log in a single Plex account and show channels from that user’s libraries. We may not implement full multi-profile support in version 1, but we will design the data model such that channel configurations could be per Plex user if needed (especially since Plex libraries differ by user).

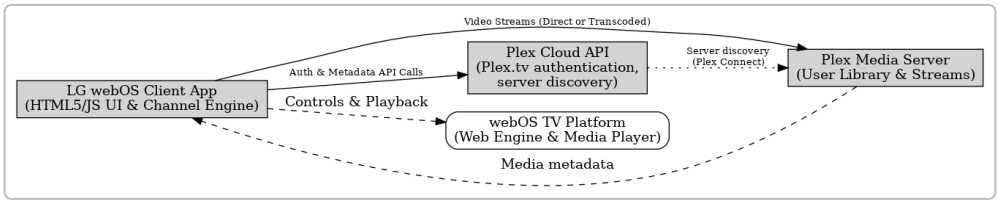
- **Settings Menu:** A settings section will allow configuration such as: selecting which Plex libraries to include in channel generation (movies vs TV shows, etc.), toggling features (e.g. “mark episodes as watched on Plex or not”), choosing streaming quality (if user wants to limit to a certain bitrate for remote streaming), UI preferences (24h vs 12h time format, etc.), and maintenance actions (like regenerating the channel lineup). QuasiTV, for example, let users pick how many episodes of a show play in a row on a channel, whether to include “specials”, and whether to shuffle episodes [20]. We can expose similar scheduling rules in settings or in an advanced channel editor interface. However, out-of-the-box, the app will provide a **zero-setup** default mode (like Coax’s philosophy of simplicity) – meaning if the user just logs in and does nothing else, a set of channels will be auto-created from their Plex content and start playing. This default could be as straightforward as “All Movies” on one

channel, and one channel per major TV genre or network for TV shows, etc., to ensure immediate fun. The user can then refine or add custom channels if desired.

In summary, the user experience goal is to **emulate traditional TV** using personal media: a laid-back, lean-back consumption model where the user can drop in and channel surf without decision fatigue. The app provides the structure (schedules, randomness, continuity) and the user's own Plex library provides the content.

Architecture Overview

Building this application involves coordinating several layers: the LG webOS client app (UI and logic), the Plex ecosystem (the user's Plex Media Server and Plex cloud services for authentication), and the webOS platform services (for media playback, etc.). The architecture is a **client-heavy approach** – all scheduling logic and UI rendering happen on the TV itself (no separate server needed for scheduling), which aligns with how apps like QuasiTV operate (QuasiTV runs on the client device like Fire TV or Android TV). Below is a high-level architecture diagram of the system components and data flows:



High-level architecture for the scheduled channels app, showing the LG webOS client's interactions with Plex. Solid lines indicate primary data flows (authentication, metadata, and streaming); dashed lines represent control and playback within the TV, and dotted lines indicate Plex's server discovery process.

As shown above, the **LG webOS Client App** (running as an HTML5/JS application on the TV) is the core of the system. It encompasses multiple sub-components:

- *UI Layer:* All the visual interfaces – the channel guide screen, video player overlay, menus, etc. – built with web technologies (we will likely use **React-based components or the Enact framework** for structure). This layer handles user input (remote key presses) and displays output (graphics, text, etc.). It will also manage focus and navigation among UI elements. The UI components will communicate with the Channel Engine and Player components to update state and respond to user actions (e.g. user presses "channel down" -> UI signals Channel Engine to switch channel).

- *Channel Scheduling Engine:* This is the brain of the "virtual channels." It is responsible for generating each channel's schedule and determining what should be playing at any given time. The engine will use Plex **metadata** (fetched via Plex APIs) to build channel lineups. For example, if Channel 1 is the "Comedy Shows" channel, the engine might compile all sitcom episodes from the Plex library, then schedule them in some order (perhaps sequentially by show, or shuffled). It will assign each episode a start time and end time on the channel timeline. Importantly, the engine must ensure schedules are **repeatable/deterministic** – given the same time and channel, it should always yield the same

content, so that the “what’s on now” is consistent. One strategy is to store a 24-hour schedule for each channel (as QuasiTV does) and update it periodically [9]. For instance, the app can initially generate 24 hours of programming for each channel. Every 15 minutes (or whenever a channel nears the end of its scheduled content), it extends the schedule by another chunk [9]. This periodic extension ensures that the schedule continuously rolls forward, but we never generate an unbounded timeline in one go (which could be unnecessary and memory-intensive). The scheduling data (list of upcoming shows with start/end times for each channel) will be stored in memory (and possibly persisted to storage so that it can be reloaded on app restart, to avoid starting over). The Channel Engine also handles selecting the correct item to play when a user tunes in: e.g. it looks at current wall-clock time, finds which media item and offset corresponds to that time on the chosen channel, and instructs the Player to start that item at the calculated timestamp. Additionally, if the user uses trick-play (pause/seek), the engine may need to adjust whether the user is “live” or behind live. It also may monitor playback events (like when an item finishes) to cue up the next item.

- *Media Player Controller:* This module interfaces with the HTML5 `<video>` element (or possibly multiple video elements for preloading, though we will likely use one at a time) to handle playback of Plex media streams. It abstracts the play/pause/seek functionality and translates remote control key events into player actions (e.g. catching a “pause” key and pausing the video). It also listens for **media events** (such as `ended` events when a video finishes, or `timeupdate` for progress) to know when to transition to the next scheduled program or to update UI elements (like progress bars). The controller might utilize webOS-specific APIs or attributes (for example, LG provides a `mediaOption` API for custom playback options and faster resume [21], and we might use `<video>` attributes or source configurations as needed). For our purposes, the standard HTML5 video element will suffice for playback, but we will set it up for optimal performance on TV (ensuring we use `preload` correctly, using media source URLs that the TV can handle, etc.).

- *Plex Integration Module:* This component handles all communication with Plex services. It will have sub-functions for **Authentication** (using Plex’s API to log in and obtain a token) and for **Data Retrieval** (querying the Plex server for library content and stream URLs). On app startup, if no token is stored, this module triggers the Plex sign-in flow (detailed later). Once authenticated, it uses Plex’s REST API (which can return JSON or XML) to fetch the user’s library structure: e.g. list of library sections (Movies, TV Shows, etc.), then within TV Shows libraries, fetch all shows and episodes. The data fetched will include metadata like duration of each episode/movie, thumbnail URLs, and so on. The module will likely use Plex’s `/library` endpoints for content and the `/playback` or `/transcode` endpoints to get actual streamable URLs for media. Importantly, when it’s time to play something, this module is used to generate the stream URL including the user’s Plex token and any parameters (like transcoding directives or offset). If the content can direct play, it will supply the direct URL to the file on the Plex server (which typically looks like `http://<plex_server_ip>:32400/library/parts/<id>.mp4?X-Plex-Token=<token>` for example). If transcoding is needed, it might request an HLS stream from Plex’s transcoder (there is an endpoint that yields an M3U8 playlist for a given item and quality). The module also handles **server discovery:** if the Plex server is remote or the user has multiple servers, it may use Plex’s cloud API to find the accessible address of the server. Plex provides a “resources” API that lists user servers and connection URLs [22] [23]. The diagram above shows Plex Cloud API helping to locate the Plex Media Server (this is the **Plex Connect** feature, wherein the Plex account can direct the app to the server’s IP/port or a relay URL if the server is not on the local network).

- *Persistent Storage:* The app will utilize local storage on the TV (through the IndexedDB or file system, or LG's `DB8` database) to save user settings and channel configurations. For example, once channels are generated, we can save their definitions (which shows belong to which channel, etc.) so that restarting the app doesn't require a completely new channel lineup generation (unless the user chooses to re-generate). Also, the Plex token and user info are stored securely (possibly encrypted) so that the user doesn't have to log in every time. LG webOS has a concept of a **"cookies"** or localStorage that persists for apps, as well as the encrypted storage for things like tokens if needed. We will follow Plex's guidelines and LG's security recommendations for storing the token (likely just using localStorage as Plex tokens are long-lived secrets).

- *webOS Platform Services:* While not a separate module in our app code, it's worth noting that the app will run within the webOS TV environment which provides services via a system bus (for things like system info, media, notifications, etc.). Our app may call webOS APIs for ancillary features: e.g. checking device capabilities (using `webOS.deviceInfo()` to see if the TV supports 4K, certain codecs, etc. for optimizing playback) [24]. Also, LG's **Media Player** pipeline is what actually decodes and renders video behind the scenes when we use the `<video>` element. We rely on it for hardware-accelerated decoding. The app might also use webOS APIs for things like preventing screensaver (there is an API or attribute to keep the screen awake during video playback), handling the **back button** properly (to exit or prompt as required [16]), and integrating with system UI (for example, if we want to expose our channels to the "Live Channels" native app – though Plex's free live channels are separate, this might be beyond scope). Additionally, the app must abide by webOS life-cycle (pause/resume when the user switches away, etc.), which we handle via webOS JavaScript APIs and our app manifest.

- *Third-Party/Open-Source Libraries:* We will leverage appropriate libraries where possible. For instance, **Enact** (LG's React-based framework) can give us a head-start on TV-optimized UI components. Enact's Sandstone UI library is specifically made for LG TVs and provides ready-made components with proper focus handling, voice control integration, etc. [25] [26]. Using Enact means our app is built on React under the hood, which is fine as long as we keep performance in check (more on this later). If not Enact, we might use standard React or another lightweight framework like Preact, combined with our own focus management solution. We may also use a library for Plex API integration; for example, the community-supported **PlexAPI** Python library (for reference) outlines endpoints, but on the client we might just directly use `fetch` or `XMLHttpRequest` to call Plex's REST endpoints with our token. No heavy backend is required since Plex itself serves the content and data. If we need to schedule content randomly or according to certain rules, we'll implement that logic in JS (potentially random seeding for deterministic random order, etc.).

Summarizing the flow: when the user launches the app, it **authenticates** with Plex (if not already logged in) via the Plex Cloud API. Once a Plex **access token** is obtained, the app discovers the Plex Server's address and fetches library info. The Channel Engine then **creates channels** (either automatically or using stored config) and schedules content using the **metadata** (durations, etc.). When the user selects a channel to watch, the app obtains the **stream URL** for the current item (with appropriate start offset) and feeds it to the **HTML5 video player**. The video plays via webOS's media pipeline. As one item ends, the next scheduled item is automatically loaded. The user can navigate the UI at any time to switch channels or view the guide; the app juggles UI and playback, pausing or stopping streams as needed when channel changes happen. All of this happens within the single webOS app context on the TV.

This architecture is entirely **client-side** except for calls to Plex services. This means the app can run offline on a local network (if Plex server is local and the user's token is cached, no ongoing internet needed except for initial auth or metadata refresh). This is similar to QuasiTV's approach – it runs on the client device and just talks to the Plex server (QuasiTV even supported running entirely on LAN for the server owner's account [27]). By not requiring a separate server for scheduling (unlike some solutions like ErsatzTV that run on a PC to generate an IPTV feed), we simplify deployment for the end-user and reduce latency (scheduling decisions are immediate in-app). The trade-off is the app must be efficient in handling scheduling logic in JavaScript, but modern JS engines on webOS (Chromium-based) and a reasonable channel count (say 10-20 channels) should be fine with proper optimization.

Technology Stack and Development Platform (LG webOS)

Web technologies (HTML5/CSS3/JavaScript) will form the basis of the app, as LG webOS TV apps are essentially web applications running on the TV's WebKit/Chromium-based engine. LG encourages use of their **Enact** framework – an open-source framework built on React, optimized for TV apps [25]. Enact (with the Sandstone UI library) provides TV-specific components (like scrollers, lists, spinners, etc.) and handles many focus management details out-of-the-box. It's also designed to perform well on limited hardware by using best practices (it ships on millions of LG TVs, so it's battle-tested) [28]. Given that our app has a significant UI component (EPG grid, menus, overlays), using Enact could accelerate development and ensure a consistent LG-styled experience. According to LG's compatibility table, Enact is supported on webOS 4.0 and above [26], which aligns perfectly with our target (webOS 4.0+). For older webOS (3.x or below), Enact isn't natively supported, but we do not intend to target those legacy TVs (and by 2025, most active LG TVs are 4.x+).

If we choose not to use Enact, we could use **React** (or even vanilla JS with a UI library). React (with something like `react-tv` or custom components) can work on webOS, but Enact essentially *is* React with added features, so there is little reason to reinvent the wheel. Another alternative is the **Lightning** framework (by Philips/TPVision, often used for high-performance TV apps with canvas rendering), but it's not officially used on webOS as much, and adopting it would add complexity (Lightning is great for animations and very heavy graphics, but our app is mostly standard UI and video, which HTML5 can handle).

Enyo (an older LG framework) is deprecated for new webOS versions [29], so we won't use that. Instead, Enact or a modern JS framework is preferred. We will also make use of standard libraries: for example, for date/time calculations (scheduling across time boundaries) we might use a library like dayjs or date-fns (small footprint) if needed, though basic JS `Date` might suffice.

App Structure: The app will be packaged as a **web app** for webOS. We have two choices: **packaged** or **hosted**. A *packaged app* bundles all resources (HTML, JS, CSS, images) into an `.ipk` that is installed on the TV [30]. A *hosted app* is essentially a lightweight stub that points to a web URL where the actual app files are hosted [31]. Hosted apps allow updating the app content on the server side without pushing new versions through LG's store each time [32]. Given that this project might undergo frequent updates (especially early on or for power users), the hosted model is attractive for agile releases. We could deploy the app on our server and update channel logic or UI quickly. The downside of hosted apps is dependence on network connectivity (if the internet is down, the app might not load content, though if the Plex server is also remote

that would be an issue regardless). Performance can also be slightly worse on first load due to network fetch of files [33] [34]. Many developers choose a hybrid approach: use hosted app during development/QA for fast iteration, then release a packaged version for the store. We will evaluate this; but since even a packaged app can fetch updated channel lineups (since content is user-driven), the need for hosted might mainly be for code updates. We might lean towards **packaged** for end users (ensuring the app loads instantly and works offline on LAN) and simply plan a robust update cycle via the LG Content Store.

Development Tools: We will use LG's **webOS TV SDK** (which includes the CLI `ares-cli` and possibly the Enact CLI if using Enact). LG provides a Visual Studio Code extension (webOS Studio) for packaging and deploying apps to the TV [35]. We will test on real LG TVs in **Developer Mode**. Developer Mode is enabled via the "Dev Mode" app on the TV, which allows us to sideload our app for testing (Dev Mode sessions last for a few hours and can be extended) [36] [37]. We'll also use the **webOS TV emulator** for quick tests, but the emulator has limitations (particularly in media playback and performance – it doesn't perfectly mirror a real device's decoders or performance characteristics [38]). Real hardware testing on various webOS versions (4.x, 5.0, 6.0, and the newer "webOS 22/23/24" which correspond to year releases) is essential. We will pay attention to differences – for example, some older webOS TVs have an older Blink engine with quirks in flexbox or ES6 features. We'll likely transpile our JS down to a safe level (Babel or similar) to ensure compatibility. The Enact build system or a custom Webpack setup will handle bundling and minification.

Media Player Technology: At the core, we have the HTML5 `<video>` element for playback. webOS's web engine supports HTML5 video, including HLS streaming, as first-class features [39] [40]. We have two approaches to feeding content: **Direct file playback** or **HLS (HTTP Live Streaming)**. Plex can provide media in either form. For local/direct play, the video element can point to a file (or a segment of a file via byte-range requests). For remote or transcoded content, Plex typically uses HLS to deliver adaptive streams. The good news is webOS **natively supports HLS** streaming [38] (meaning we can give the video element an `.m3u8` URL and it will handle the manifest and chunk downloads internally). We should verify which HLS features are supported: per LG's docs, certain advanced tags (like `EXT-X-PROGRAM-DATE-TIME`, used to timestamp segments, or `EXT-X-DATERANGE`) are *not* supported on webOS [41]. Notably, `EXT-X-PROGRAM-DATE-TIME` is not supported [42], which means we cannot rely on that tag to synchronize to wall-clock (if we had considered aligning channel streams to real time via HLS manifests – an idea we had but the client-side scheduling renders it unnecessary). Also, discontinuities in HLS (for stitching together different videos) are only partially supported (PTS discontinuities are okay on webOS 4.0+, but not codec changes) [43]. This suggests that trying to create one monolithic HLS stream that stitches multiple Plex items might be problematic. Instead, the app itself will handle transitions by ending one `<video>` source and starting the next. A brief black screen or loading spinner between programs is acceptable (much like a quick channel fade-out/fade-in).

Supported Media Formats: LG webOS TVs (especially 4.x and above) support a wide range of codecs: H.264, H.265/HEVC, MPEG-2, VP9, and even newer ones like AV1 on recent models [44] [45]. Most Plex libraries will contain H.264 or HEVC content in MP4 or MKV, which is well within webOS capabilities [45]. Audio codecs like AAC, Dolby Digital (AC3/EC3), MP3 are supported; DTS is listed for some containers [46] (though some newer LG TVs dropped DTS decoding due to licensing – it's listed as supported in the dev docs for MKV, but in practice models after 2020 might not play DTS). We should be mindful: if a Plex video has DTS audio, the TV may not produce sound unless the user has a receiver that can decode it. We might prefer to ask Plex for an AAC or AC3 audio track in that case (Plex's transcoder could downmix DTS to AC3 on the fly if needed). This is part of the **transcoding logic**: only transcode when needed. We'll leverage Plex's decision-making as much as possible: Plex has a notion of client capabilities (via the client profile). We can advertise a client

profile to the Plex server so that it knows our app can direct-play certain codecs and will request transcodes for others. According to QuasiTV's notes, they added a "default transcoding profile which should only transcode video if necessary" in an update [3] – we will adopt a similar stance: direct play unless absolutely not possible. In particular, webOS can't do 10-bit H264 or VC-1, etc., so those might need transcode; also very high bitrate 4K might need downscaling if the network is insufficient. We can expose a setting (e.g. "Prefer Transcoding = Auto/Direct Only") to users with advanced needs.

Video Player Implementation: We will create a single `<video>` element in our application's DOM (potentially full-screen behind the UI, Z-ordered such that UI overlays can appear above it for OSD). When the user changes channels or when one program ends and another begins, we have a few implementation options: 1. **Sequential Source Switching:** Easiest method – use the same `<video>` element. When it's time to play something new, call `videoElement.pause()`, then set `videoElement.src` to the new media URL, then call `play()`. For seeking on startup (if starting mid-program), we will attach an event listener for the `loadedmetadata` event (or use the promise from `video.play()` on modern browsers) then set `video.currentTime = offsetSeconds`. We should also consider using the `MediaSource Extensions (MSE)` API if needed for more fine-grained control, but likely not necessary for our use case. MSE would allow us to append media segments from multiple files into one stream buffer (for a seamless transition), but managing that for dozens of shows (each potentially with different encodings) would be complex and, given HLS discontinuity issues, it might not yield a better result. The slight pause between items is acceptable and mimics a channel ad-break or transition. 2. **Preloading technique:** If channel switching needs to be extremely fast, we could instantiate a second `<video>` element off-screen and load the next channel's stream, then swap it in. However, since we don't know which channel the user will switch to ahead of time (unless we assume sequential surfing and try to preload the adjacent channel which might be a nice optimization), this is of limited benefit and doubles resource usage. So we won't do multiple concurrent video decoders except possibly buffering a few seconds of the upcoming program on the same channel if we could predict it – but again, not necessary because the content is local network or similar, so rebuffering is quick. 3. **mediaOption for Quick Resume:** LG provides a `mediaOption` parameter that can speed up starting playback if quickly resuming the same content [21], but in our case we are usually switching to different content, so it might not apply. However, if a user is frequently toggling between two channels (channel surfing), we might consider leaving a short buffer or not immediately dropping the connection of the last channel for a couple of seconds to allow quick back-and-forth. This is an advanced optimization and likely unnecessary initially.

We will ensure to handle the **video element lifecycle** properly. For example, when stopping playback, call `videoElement.removeAttribute('src')` and `videoElement.load()` to release decoder resources (especially if not immediately playing something else). This helps avoid memory leaks of video decoders on some platforms. Also, we need to hide the video element when not in use or when UI menus are fully covering (some apps do that to reduce GPU load, but we probably will keep it showing unless user goes to a pure menu).

LG webOS Specifics: We'll use the **webOS JavaScript API** (through the `webOSTV.js` library provided by LG) for tasks like retrieving device info, system state, and maybe handling certain system events. For instance, if our app is running and the user hits the physical **Home** button on the remote, our app will be paused – we should listen for the `visibilitychange` event or LG's lifecycle events and pause the schedule updates, then resume when back in focus. If the app is backgrounded for too long or the TV goes to screensaver, we might refresh the schedule on resume to avoid drift.

External Dependencies: Aside from Plex, our app doesn't require external cloud services. All data comes from the user's Plex. We should however be prepared to handle Plex authentication via internet (Plex token retrieval needs plex.tv). If the TV is offline or Plex is unreachable, our app should gracefully inform the user. (Perhaps show an error "Cannot reach Plex server" and allow them to retry or adjust settings.)

Table: Key Technologies and Rationale

Layer/Component	Technology Choices	Rationale and Notes
UI Framework	Enact (React) or React + custom	Enact [26] provides TV-optimized components (Sandstone theme) and is officially supported on webOS 4.0+. This accelerates development of focus management, lists, etc. React (with JSX) enables modular UI. We will follow LG's UI guidelines for TV for consistency.
State Management	Redux or Context API (if using React)	The app has multiple interacting parts (guide data, player state, etc.). A predictable state container like Redux can help manage this complexity. However, to avoid performance issues, we'll keep state updates granular and possibly use Context for simpler global data. We'll memoize and optimize re-renders (as recommended for TV apps [47]) to maintain 60fps UI.
Video Playback	HTML5 `<video>` element (built-in player)	Leverages hardware decoding on TV. Supports HLS and common codecs. Simpler than integrating an external player. We may use MSE/EME if DRM was needed, but Plex personal media is usually unencrypted. The `<video>` element will handle audio/video sync and scaling. We'll ensure to use proper attributes (e.g. `playsinline`, etc.) as needed.
Plex API Comms	Fetch/XHR with Plex REST API (JSON/XML)	No official Plex SDK for web, but the HTTP API is well-documented by community. We'll craft requests including the required headers (`X-Plex-Token`, `X-Plex-Client-Identifier`, `X-Plex-Product` etc.) as per Plex API spec. We might build a small wrapper class to call endpoints like `/library/sections`, `/library/metadata/{id}`, and `/players/{id}/playback` if needed. Authentication will use Plex's pin flow or the user can manually input credentials/token.

Layer/Component	Technology Choices	Rationale and Notes
Data Storage	IndexedDB or `localStorage`	For caching channel configurations and the Plex token. IndexedDB is asynchronous and good for structured data (e.g. we can store channel lineup JSON). `localStorage` is simpler and might suffice for small data (token, last used server). For security, sensitive data (token) should not be exposed – Plex tokens are long but if someone got it they'd have your library access, so we'll treat it as we would a password (not logging it, etc.).
Styling	CSS3 (Flexbox, grid) with responsive design	The app will run at 1080p or 4K resolution depending on the TV. We will likely design for 1080p as base (since many UI coordinates in webOS are for 1920x1080, and on 4K TVs the system might scale or we can explicitly handle higher resolution). Using flexbox or CSS grid will help layout the EPG grid and other components dynamically. We will be cautious of performance – excessive CSS shadows or filters can hurt performance on TV. We'll stick to simple, GPU-accelerated transforms for animations (like focus scaling).
Internationalization	Possibly support localization (multi-language)	Not core for MVP, but Enact has i18n support and we might use LG's recommended i18n library. Plex metadata (titles, etc.) is usually in the user's language if their media is titled as such. UI labels we will keep easy to translate.
Logging/Debugging	Console logging (removed in prod), DevMode logging	During development, we'll use `console.log` generously (the Dev Mode app on the TV allows connecting Chrome DevTools to the TV app for debugging, which we will use). We might integrate an on-screen debug menu or log overlay that can be toggled in Dev mode to see channel calculations or errors (but obviously not in production).

This tech stack ensures we align with **LG's platform capabilities** and use modern, maintainable tools. The key is to keep the app efficient: webOS's browser engine is not as fast as a PC's – it's often an older Chrome version and runs on limited CPU/GPU. By using frameworks like Enact and following best practices (minimize reflows, use requestAnimationFrame for any animations, avoid heavy computation on the main thread), we can achieve a fluid experience.

We also note that LG's web engine might not support some newer web APIs – we will check the **Web API support charts** LG provides [48]. For example, if we plan to use certain ES2020 features, we ensure the engine's version (for webOS 5.0, likely Chrome 68; for webOS 6.0, Chrome 79; newer webOS 22 might have a Chromium 87+). We'll transpile code to ES6 for broad compatibility and include polyfills if needed (especially for things like `fetch` on older versions or Promise if not present on webOS4 – though it likely is).

In conclusion, our stack will be: **Front-end:** Enact/React JS app; **Player:** native HTML5 video with HLS and MP4; **APIs:** Plex REST + webOS JS API; **Storage:** local (on-device) for config; **Build/Deploy:** LG webOS packaging, potential hosted deployment for rapid updates. This leverages LG's recommended approach and Plex's flexible API to deliver the required functionality.

Media Playback Design and Optimization on webOS

Smooth and reliable media playback is central to this app's success. We must ensure that when a user flips to a channel or when a scheduled item starts, the video plays promptly and with minimal buffering. Below we address media format support, streaming approaches, and how we handle seeking and transitions, all tailored to LG webOS capabilities:

- **Supported Formats & Direct Play:** LG webOS TVs support a wide range of video codecs/container formats natively, which covers most Plex media without transcoding. For example, webOS 6.0 devices support H.264 up to 1080p60 and 4K30, HEVC up to 4K60, VP9 up to 4K60, and even AV1 up to 8K on certain models [44] [49]. Common file containers like `.mp4`, `.mkv`, `.ts` are supported [45]. We will retrieve the media's codec information from Plex and decide if the TV can play it directly. If yes, we instruct Plex to **Direct Play** (i.e. we take the direct URL to the media file on the server). This is efficient as it uses no server CPU and typically delivers the full quality stream. We do need to ensure the Plex server will allow direct stream – usually if the client is on the same network or if remote with appropriate settings. We include the Plex token in the URL to authenticate.

- **Transcoding & HLS:** If a media is not compatible (e.g. a high 4K profile that the TV can't decode, or an audio codec not supported, or perhaps the user's network can't handle the original bitrate remotely), we'll fall back to **transcoded streaming**. Plex's transcoder can output an HLS stream with segments encoded in H.264 or HEVC at a chosen bitrate/resolution. We will set up a **transcode profile** that matches the TV's capabilities: for example, if transcoding, prefer H.264 Level 4.2 or HEVC Main profile, AAC audio – all of which webOS can play. The only channel format Plex clients truly universally support via HLS is MPEG-TS segments (they noted MPEG-TS in HLS is a common denominator) [50], but modern webOS can also do fMP4 segments. We might not need to worry that deeply – Plex will serve something compatible if we identify as an LG TV client. We can send `X-Plex-Product` and `X-Plex-Client-Profile` headers to guide the server. According to LG docs, HLS is fully supported on device (the emulator had limited support before 5.0, but on actual TVs it works) [38]. So, when a transcode is required, our app would get an HLS playlist URL from Plex (possibly via the `/transcode` endpoint or by creating a **PlayQueue** and then calling Plex's timeline player – however, a simpler way: Plex has a mechanism where if you append certain query params to the media URL, the server may produce an HLS stream. Alternatively, we might use the Plex HTTP Live Streaming endpoint directly specifying desired params). Once we have the `.m3u8` URL, we set `video.src = thatUrl`. The TV will then fetch segments and play. The app can still seek within that stream (seeking will cause the player to request the appropriate segment near the new time).

- **Initial Buffering and Seek Behavior:** On tuning a channel, we want to minimize the time to play. The `<video>` element's default behavior on setting a src is to buffer a bit then play. We will use the

`video.play()` promise or events to know when playback begins. If direct playing a large file, one consideration is **start offset**: if we need to start 1 hour into a movie, we can do one of two things:

- Use the `Range` header to ask the server for bytes from a certain offset (this is behind the scenes – the browser might handle it when we do `video.currentTime = X`). We should confirm the media file has fast seek (MP4 with moov atom at front, etc.). Most Plex-optimized files do.
- Or use HLS even for direct play scenario by asking Plex to generate an HLS with a start time. Plex's transcoder can generate a stream that starts at a given time offset of the media. But doing a full transcode for that is overkill if the format is actually playable. So better to direct play and then seek via the player.

The plan: **Direct Play Seek** – we set the video src to the direct media URL (which starts from beginning), then immediately (or on metadata loaded) set currentTime = offsetSeconds. This will trigger the internal buffering at that time. The LG player supports standard seek operations [51] ("Seek: Supported", including for live contexts from webOS3.0+ [40]). We note that "Fast Forward & Reverse: Not supported" in LG docs [40] – this means the video element doesn't support changing playback rate >1 or reverse playback; however, we will implement FF/RW as discrete skip steps (e.g. skip 10s forward on each press) rather than continuous fast-forward. That is acceptable UX for a channel.

We will verify that when performing an immediate seek on load, the player doesn't stall. Usually, one trick is to wait for `loadeddata` or a small time delay before seeking, but often setting currentTime in the `loadedmetadata` event is fine. Another approach: use the media fragment URI if supported (like `video.mp4#t=3600` to start at 1 hour). Not sure if webOS player respects that – if it does, that's the simplest (but it's an older HTML5 spec that not all players fully implement). We'll likely do manual seek.

- **Playback Continuity:** When one item ends, the Channel Engine will pick the next item and we perform another source switch. We'll hide any momentary loading. We could possibly overlap the end of one and start of next by a second to avoid any gap (but syncing that is tricky, we may accept a second of black or slight overlap if we preloaded a second video tag). Given typical user expectations, a quick black screen between shows is fine (like how Pluto TV or live channels often have an ad or bumper; we could even display a "Up Next: [Title]" for 1 second to mask the load time). The LG player's **discontinuity support** is limited [43] , so we won't try to make a single continuous stream of multiple shows.

- **Audio & Subtitles:** We will allow enabling subtitles if available. Plex can supply subtitles either as separate files or embedded. The HTML5 player on webOS supports WebVTT subtitles natively [52] . If Plex provides SRT, we might need to convert to WebVTT on the fly or perhaps Plex can burn them in via transcoder. Ideally, for any serious watching with subtitles, the user might just treat it like normal playback – but since our aim is more "lean-back", we might not emphasize subtitles (like a TV broadcast, you either have closed captions or not). However, we see QuasiTV had a subtitle toggle added in version 0.6 [53] . We can do similarly: if the user hits a "CC" button, and if a subtitle track is available for that media, we enable it. Implementation: if direct playing an MKV with embedded subs, the video element might not automatically show them; we might have to use `<track>` elements or Plex might provide a separate URL for the subtitle file (which we can fetch and add as a text track). This is a detail we'll handle but it's not core to the linear experience.

- **Performance Considerations for Video:** On webOS, rendering a video at 1080p or 4K is hardware-accelerated. We must avoid having too many overlapping elements on top of video that force composition in software. Using CSS animations or translucent overlays is fine (the hardware compositor can handle a video layer + one UI layer). But something like a huge CSS blur over video would be bad. We'll keep overlays simple (mostly text and small images). We'll test on lower-end models to ensure video playback stays smooth.

- **Edge Cases:** We should handle if a video fails to play (maybe a file is missing or a network hiccup). In that case, we can attempt a retry or skip to the next item after a short error message. QuasiTV implemented a retry on error only if the video had played less than 20 seconds [54] to avoid looping on a corrupt file – we can incorporate a similar rule. If the Plex server is unreachable mid-way, we'll notify the user and perhaps auto-retry connecting periodically.

- **Live Seek and Time-Shift:** Because our channels are pseudo-live, the concept of "live seek" (like going back to an earlier point in the current program) is effectively like DVR. LG's platform supports live seeking (i.e. seeking in a live stream buffer) on webOS 3.0+ [40]. In our implementation, since each program is effectively a separate video stream, our "buffer" doesn't naturally extend beyond that program. If we allowed going back into a previous program that has already ended, we'd have to load that video anew – which complicates things and breaks the live paradigm. So, likely we will restrict seeking to within the currently playing item. The user can rewind say up to the start of the episode or up to X minutes (if we limit it). If they try to rewind beyond it, we either stop at the beginning or perhaps load the previous program's video on demand (but then they are in the past and we'd need an interface to jump out – not worth it for now).

- **Testing Formats:** We will test direct play with high-bitrate content to ensure webOS can handle it (some older TVs might choke on very high bitrate 4K even if theoretically supported, due to I/O or decoder limits). If issues are found, we may impose a default max bitrate (like 20 Mbps) for remote streams and let users override if needed.

In summary, our media playback strategy is **Direct Play first**, using native formats, with **smart seeking** to simulate joining in-progress, and fallback to **transcoded HLS** when required. We leverage the TV's native support for HLS so we don't need heavy logic to manage segments ourselves, which simplifies development and relies on proven LG internals. We will keep transitions simple (stop one stream, start the next) and maintain an illusion of continuous playback through UI cues rather than complex continuous streaming tech, given webOS's partial HLS discontinuity support. This approach prioritizes reliability – fewer moving parts – and performance, since each `<video>` playback is a fresh start that the platform is well-optimized to handle.

Plex Integration and Communication

Integrating with Plex is a multi-faceted challenge: user authentication, data fetching (libraries and metadata), and stream URL generation with proper handling of direct play vs transcoding. We will utilize Plex's public (but not extensively documented) APIs and follow patterns established by third-party Plex clients.

User Authentication: To access a user's Plex content, we need their Plex **authentication token** (often called `X-Plex-Token`). Plex does not provide an OAuth for third-party TVs but does have a device pin-based auth mechanism. The plan is to implement the **PIN claim authentication flow** [55] for a smooth TV login experience: 1. On first launch (or if the user is not logged in), the app will generate a new Plex PIN via a POST to `https://plex.tv/api/v2/pins` with our app's client identifier and product name [56]. Plex responds with a `pin ID` and a short `code` [57]. 2. We then direct the user to authenticate. Since typing a full username/password with a TV remote is tedious, we prefer the **second-screen approach**. We will display on the TV a screen with an instruction: "To link Plex, visit plex.tv/link on your phone or PC and enter code: ABCD". The `code` we got from Plex is typically 8 characters (mix of letters/numbers) which the user can input on plex.tv/link. Alternatively, Plex's API documentation suggests constructing an **Auth URL** that includes the code and can be opened in a web browser [58] – we could show a QR code with that URL to make it even easier (user scans QR and logs in on their phone). 3. Meanwhile, our app will **poll Plex** for the PIN status (calling `GET https://plex.tv/api/v2/pins/<PIN_ID>` every few seconds) [59] [60]. When the user successfully enters the code on plex.tv, Plex will mark the PIN as claimed and attach an `authToken` (the user's Plex token) to that PIN record [61]. Our app will receive this in the polling response. 4. We then store that token securely and consider the user "logged in". From then on, all Plex requests will include `X-Plex-Token: <token>` in the header or as a query param.

This PIN flow is exactly how official Plex clients on devices like Apple TV or Roku do linking. It's user-friendly and avoids exposing credentials to our app. (As a fallback, we might allow the user to manually enter their Plex username and password in a pinch – and then call Plex's `/users/sign_in.json` – but this is not preferred, and Plex tokens last basically indefinitely so linking is better.)

We will also send identifying headers in each Plex API call, as Plex requests: e.g. `X-Plex-Client-Identifier` (a UUID we generate and keep for our app instance) [22], `X-Plex-Product` (app name), `X-Plex-Version` (our app version), and possibly `X-Plex-Device` (like "LG webOS TV") and `X-Plex-Platform` ("webOS"). This ensures Plex knows what client we are (and it will show up in the user's Plex account devices list as such).

Server Discovery: Once authenticated, we must find the user's Plex Media Server (PMS). If the PMS is on the same network (common case for a personal Plex), we could attempt to auto-discover via UDP (Plex uses GDM multicast for discovery) but doing that in a webOS app might be impossible due to sandboxing. Instead, the standard way is to call `https://plex.tv/api/resources?includeHttps=1&X-Plex-Token=<token>`. Plex will return an XML (or JSON if requested) listing all servers and clients linked to the user, along with their connectivity info (IP addresses, port, whether they're local to the current IP, and relay URLs if outside) – this is often used by mobile Plex clients. We will parse that and pick the server that matches the one we want (if the user has only one PMS, that's it; if multiple, and we haven't chosen, we might list them for the user). Once we have the server's base URL (e.g. `https://<ip>:32400` or a relay like `https://xxxxx.plex.direct:32400` which includes certificate pinning), we will use that for subsequent library queries. QuasiTV also added support for selecting from multiple servers in v1.4 [62], so similarly we'll handle multi-server if needed.

Fetching Library Content: With the server URL and token, we can retrieve library data. Key steps: - Get the list of library sections (`/library/sections`). This tells us the sections (like "Movies", "TV Shows", etc.) and their types. We likely only care about sections of type "show" and "movie" because we won't schedule music or photos. - For each relevant section, we can fetch the items. For a "TV Shows" section, hitting `/library/`

`sections/<ID>/all` yields all shows. Each show entry includes metadata and an ID. We might then fetch `/library/metadata/<showID>/allLeaves` to get all episodes, or step by step: first seasons then episodes. But Plex has handy library endpoints: e.g. to get all episodes of all shows at once might be heavy, so instead we might do it show by show as needed (especially if the user has a huge library, we might not want to load every episode upfront). We can optimize by delaying full episode retrieval until needed by scheduling (like we schedule maybe one show at a time and only pull episodes for shows that are on channels). However, since our goal is to auto-create channels across the whole library, we likely will need to gather a lot. Perhaps an approach: for each show we add to a channel, retrieve its episode list sorted by episode index. Or we can use Plex's smart filtering: Plex does allow queries like `/library/sections/X/all?type=4&genre=Comedy` etc. But those are not super documented. Simpler: fetch all shows, filter by some criteria in our app.

- **Channel generation logic (content selection):** The app may have preset algorithms: e.g. automatically create channels by **studio/network** (like QuasiTV's premade channels which grouped shows by their original network [9]), or by genre, or just a catch-all. Initially, we can do something straightforward: one channel for each *top 5 genres* in the library, etc. The user can always edit later. If the user has a playlist or collection (like "Kids Movies"), we might allow channel from that (QuasiTV 2.1 added that feature [10]). Implementation-wise, for automatic grouping by studio, we might rely on metadata: Plex's metadata for shows has a "studio" field (the network). QuasiTV had to adapt when Plex changed their agent and the studio field's meaning changed [63]. They ended up retrieving the "network" differently via an API. We might avoid that complexity at first and just group all shows from say the same genre or we make a single channel that includes all shows (shuffled).

Regardless, once we decide what content goes into a channel, we need to retrieve all items for that channel. For example, Channel "Comedy" might include 10 shows – we gather all episodes from those 10 shows. We then perhaps shuffle them or schedule sequentially by show. QuasiTV schedules shows in episode order but may rotate through shows in blocks [9]. We can emulate: e.g. pick a random start episode for each show so that not all channels start at S1E1 (QuasiTV did a random offset to avoid uniformity [64]). Then schedule perhaps 1-3 episodes of that show, then move to next show. The scheduling engine will make these decisions, but the Plex integration part is providing the data (episode durations and order).

We'll store media metadata in memory for quick reference – e.g. a dictionary of episode lengths by episode ID, etc., after initial fetch.

- **Playback Launch (Getting Streams):** When it's time to play a specific item (say Episode 5 of *Friends*), how do we get a playable URL? Plex has a few ways:
- The item metadata from Plex includes a `<Media>` element with `<Part>` children that contain file information. Usually, the `Part` has an `id` and a key (like `/library/parts/<partId>.mp4`). If we take `http://server:32400<Part key>?X-Plex-Token=...`, we can retrieve the file directly. We need to confirm if we need to append `&download=1` or some parameter to force direct download – but I think for a local Plex that URL will stream the file data. The Plex token ensures permission.
- Alternatively, Plex provides a `/player/playback/playMedia` API (POST) that official clients use to instruct a Plex player to play something, but in our case we *are* the player, so we don't need that.
- For transcoding, Plex's URL can be constructed like: `http://server:32400/video/:/transcode/universal/start.m3u8?...` with numerous parameters (like `mediaIndex=0`, `partIndex=0`, `maxResolution=..., bitrate=...` etc.,

plus a session ID and token). We might use a simpler approach: Plex's `/playlist.m3u8` URL. There is an undocumented one: `/library/metadata/<id>.m3u8` which can give a stream if the client profile is known, but not sure if that's official. Alternatively, we use the `resources` approach: For remote play, Plex might require going through a relay which is already given in the resource list with a specific hostname (plex.direct with certificate). We should ensure any URL we use is accessible. Plex uses TLS with custom cert for remote (via plex.direct domain). Our app should be fine using those since webOS supports HTTPS (we saw note: HTTP/2 supported from certain versions, HTTP/3 not, but that's okay) [38] .

Given complexity, we will likely use **Direct play URL** for direct plays and **transcoder API** for forced transcodes. We can detect if transcode is needed by comparing the media's codec/profile to our supported list or by a simpler route: attempt direct play, and if the player errors out quickly (e.g. codec not supported error), then switch to transcode. QuasiTV initially didn't support transcode and would just error if a file was incompatible; they later added a default profile to transcode video if needed [3] . We can be proactive: maintain a list of unsupported formats (like if a video is VC-1 codec, we know webOS likely can't play that in an MP4 container – though doc says VC-1 in ASF is supported except WMVA [65] , but if it's MKV with VC-1, possibly not). Also older TVs can't do Hi10p anime encodes – those definitely require transcode. We'll identify those via Plex metadata (which gives codec and perhaps an "decision" field if we query with a client profile).

Another potential method is to use the **Plex Web API** (the one the Plex web app uses). For example, we could open a websocket to PMS and use their timeline protocol to mark when we start playing something and at what position (this is how Plex syncs play states). If we want, we could report playback status to Plex (like "now playing on device X"). QuasiTV did implement an option to report play progress but then reset it to avoid marking watched [66] . We can do similarly: using the `/:/timeline` endpoint to POST play progress. This would allow the user's Plex account to know what they've "watched", but since the idea is passive watching, many might not want their content marked as watched. QuasiTV by default resets progress at end so Plex doesn't consider it watched [66] . We will do the same: maybe update the timeline as playing (so if the user opens Plex on another device they see "Playing on LG TV: The Office S2E5 at 10:15"), but at completion we either mark it unplayed or only scrobble if user explicitly wants. This is more of a user preference; we'll likely default to *don't mark as watched*. In any case, timeline reporting would be a nice-to-have to integrate with Plex's ecosystem but not strictly needed.

Metadata Display: For each item, we will use Plex's provided metadata: titles, episode summaries, posters/thumbnails. Plex APIs return URLs for images (e.g. `/library/metadata/<id>/thumb` or the newer `photoTranscode` endpoint to get a specific sized image). We will use those to show images in the guide or info panels. Since our app is hosted on the TV, and the Plex server might be local, loading images should be quick. We'll just ensure to include the token in those URLs too. We might consider caching posters to disk or memory to avoid repeatedly fetching them (though Plex images are usually cached by the TV's browser if we use stable URLs).

Error Handling and Resilience: If the Plex token expires or becomes invalid (unlikely, as Plex tokens don't expire unless user revokes), we'll catch 401 responses and prompt re-login. If the Plex server is unreachable, we'll show a message "Plex server not found – please check connection" and perhaps provide a retry. We can also ping the server occasionally (or rely on errors when fetching/streaming).

Multi-Server or Shared Libraries: Some users have multiple Plex servers or access friends' libraries. We can list available servers (from the resources API) and let the user pick which one to use for channels (or even combine content from multiple? That might be complicated in terms of tokens and different hosts, so probably one server at a time). We'll stick to one server context per session.

Plex API Rate Considerations: We should be mindful not to overload the Plex API with too many calls. We can cache library lists on first load. If libraries update (new content added), the user might need to refresh or a daily refresh could occur. Plex does send notifications via websocket to clients for library changes, but implementing that might be overkill. Simpler: provide a "Refresh library data" action if needed or refresh at app start.

In essence, the integration will make our app appear as a Plex client to the server. By following Plex's guidelines for token auth and using the token in all requests, we can do anything Plex's own app can, within the bounds of Plex's API. The approach is **stateless** in terms of each request (since token is provided each time). We'll keep the Plex token in memory and maybe a short-term cache of some key data. The heavy lifting (transcoding etc.) is done by Plex when asked; our client just renders the final stream.

Channel Scheduling Engine Design

The channel scheduling engine is what differentiates this app from a simple media browser. Its job is to create a *deterministic, linear schedule* for each channel and to serve up the right content at the right time. We'll outline how to design this engine for both **determinism** and **flexibility**, ensuring that if you tune in at a given time, you consistently get the same program (given the same library and settings), and also that channel content doesn't repeat too frequently unless intended.

Schedule Representation: We will model the schedule as a timeline for each channel, divided into contiguous segments where each segment corresponds to one media item (episode or movie) with a start and end timestamp (relative to a continuous timeline, likely wall-clock time). We can choose a reference point for time zero – a simple choice is to align schedules to the real-world clock. For example, we could decide that Channel 1's schedule "day" starts at 00:00 (midnight) each day in the user's local time. However, since content lengths are irregular, we won't align strictly to top-of-hour or half-hour boundaries like broadcast TV might. Instead, each program will start immediately after the previous one ends. This means our EPG will show non-uniform block lengths; that's fine, as seen in QuasiTV's guide (some blocks end at :22 past, etc., reflecting actual durations).

To ensure determinism, we might anchor the schedule to a fixed point in time – e.g., the day the channel was created. QuasiTV appears to maintain a persistent schedule that updates over time [9] . They have a concept of keeping a 24h schedule updated every 15 minutes, which implies they slide the window forward. We can do similarly: - Generate schedule from "now" for the next 24 hours. - Store it (in memory, and optionally persist to disk for recovery). - As time moves, say every 15 minutes or when a program ends, extend the schedule further to always have some buffer of upcoming content scheduled. - Optionally prune past entries to keep the data structure manageable (we don't need to store last week's schedule except maybe for reference if implementing rewind beyond current program).

However, if the app restarts, how do we ensure the new schedule aligns with the old? One idea: use a pseudo-random number generator with a fixed seed per channel to decide content ordering, combined with a known start time. Alternatively, store the last known point and resume. A simpler approach: we persist enough info to rebuild deterministically. For example, for each channel store: the last scheduled item and how far in it was at app close, or store the "channel clock" which is essentially an offset of the channel's timeline vs real time. If at app close, Channel 1 was playing Episode X at 50% in at real 10:00pm, and the channel's theoretical schedule started at some reference, we can on restart align the timeline such that at the current time it would have been at that progress (assuming continuity). Actually, easier: we can persist the "current item index and offset" for each channel at a known timestamp (like at app exit or periodically). Then at app launch, recalc that by fast-forwarding from that state by the elapsed time while app was off. This ensures channels don't "restart" fresh each app launch – they pick up as if they were running in the interim.

For a first version, we might cut a corner: assume the app is running continuously. If the app restarts, just regenerate channels anew (which could mean content might differ after a reboot). This isn't ideal, but given that it's a personal media scenario, it might be tolerable. We'll aim to minimize restarts (the app can be left open). But to be robust, we'll implement some persistence.

Deterministic Scheduling: We want the schedule to not shift randomly each time. So our scheduling algorithm will be rule-based rather than on-the-fly random. For example: - For TV episodes: We could maintain a per-show pointer for where we left off in that show's episode list for each channel. Then when scheduling that show next time, continue from that episode. QuasiTV's dev mentioned "QuasiTV knows the last scheduled episode per show per channel… so it just continues from there" [67]. This means if a channel cycles through a set of series, it doesn't always start those series at episode 1 every rotation – it picks up where it last stopped. Over time, this results in eventually airing all episodes. We will incorporate this logic: keep track of episode progress for each series in each channel. - For Movies: If a channel is a random mix of movies, we might want to avoid repeating the same movie too often. We can maintain a history of what was recently played on that channel and try to not repeat until a certain quota of others have played. Essentially, implement a "shuffle" without immediate repeats (like a deck of cards – don't reshuffle until all have played, if possible). - Blocks: We might allow channels to play a few episodes of the same show back-to-back (for viewer comfort). E.g., if Channel 5 is the Sci-Fi channel with Star Trek and Star Wars films and series, maybe it plays two Star Trek episodes in a row before switching to a different show. This block length could be configurable (QuasiTV allowed 1-4 episodes in a block as a setting [68]). Implementing that: when scheduling, if the next show in rotation has more episodes available, schedule 1-4 sequential episodes from it, then move to next show. - **Time slots vs. continuous:** We are not doing fixed timeslots (like not forcing new show at exactly 9:00pm) because our content lengths vary. This is simpler and maximizes content usage. The downside is the guide doesn't have neat alignment, but that's fine.

Channel Clock and Start Offsets: To avoid monotony (all channels starting their first show at exactly the same time point in that show), it's good to offset the schedules initially. Coax's developer noted that in his app, "it loads up movies at random parts and feels very much like channel surfing" [7]. This suggests Coax might intentionally start channels at a random offset in a movie so that when you first use it, you truly see mid-scenes, enhancing the live feel. We can replicate that: at channel creation, for each channel, randomly choose a start point in the scheduled content timeline. For example, say Channel 2's lineup (if played from the beginning) would start at *Movie A*. Instead, we might start it in the middle of *Movie A* so the first time you tune it's halfway through. That offset can be random per channel but then fixed afterwards. It's essentially defining that channel's alignment to wall-clock. We'll store that offset (or initial item and position). Then all

future scheduling is relative to that. This "math and your imagination" approach (to quote the Coax dev [69]) means the app doesn't actually run all channels in background (wasting resources); it just uses math to simulate that they were running.

Algorithm for Building Schedule: Pseudo-code for initial schedule build of one channel:

```
channelItems = [list of media items for channel in desired order cycle]
if channel is TV shows grouped, channelItems might be segments like "show1
episode1-10, show2 ep1-8, ...".
determine channelStartOffset = random(0, total_duration(channelItems))   // pick
a start point in the continuous loop
currentTime = now (e.g. epoch ms or minutes of day)
scheduleStartTime = currentTime - (channelStartOffset mod total_duration)   //
align so that the offset from start corresponds to now
timeline = []
timePointer = scheduleStartTime
while timePointer < scheduleStartTime + 24h + safetyBuffer:
    for item in channelItems (in cycle order):
        duration = item.duration
        if timePointer + duration < currentTime - pastBuffer:
            // skip scheduling items that end before "now" if we are starting
from an offset
            timePointer += duration
            continue
        start = timePointer
        end = timePointer + duration
        timeline.append({item, start, end})
        timePointer = end
        if timePointer > scheduleStartTime + 24h:
            break out once we've got 24h ahead
```

This rough idea would create a rolling schedule. In practice, we might not explicitly cycle a fixed list infinitely; we might instead generate sequentially and loop back as needed. We also incorporate real-time updates: if now moves, as long as our timeline covers now to now+24h, we're good. Every so often, as now shifts, we append more.

We will also handle boundaries: when we reach the end of a series, either loop back to beginning or mark that as a point to maybe switch series if it's a multi-show channel. We can treat multi-show channels as interleaving episodes from multiple shows. QuasiTV's initial mode was channel per studio, which effectively grouped many shows – they just put them one after another in alphabetical or network order in blocks [9]. We might adopt a simpler approach first: one channel = one show (like a 24/7 marathon of that show) or one channel = shuffle of all episodes from a set of shows. The complexity of scheduling multiple series in rotation is doable but requires the tracking per series of last episode, etc.

Real-time Behavior: The engine will have a loop or use timers to monitor the playback: - When a video is within e.g. 1 minute of ending, prepare the next video's info (so the UI can show "Next: X"). When it actually

ends, trigger the player to load next. - If user changes channel, we calculate the current item on that target channel by looking at our schedule data for that channel with current time. - We should ensure calculations account for time precisely – e.g. if an episode ends at 10:00:37 PM, and the user flips at 10:00:40, it should already consider the next episode started at 10:00:37 (with 3 seconds elapsed). Our schedule data gives us that.

Handling Overflows: If our schedule runs out (which it shouldn't if we keep extending), we extend. If an extension falls on a day boundary or a new loop, fine. If content library changes (new episodes added), we might want to incorporate them. Possibly we refresh channel content daily to add new episodes that might have been missing. Not critical but a nice future addition.

User Interaction with Schedule: If user scrolls the guide far into future, we may need to generate further out schedule on-demand so they can keep scrolling. QuasiTV allowed scrolling until the end of scheduled content, which is finite (if they do only 24h ahead, you can't scroll past 24h) [3]. We'll likely cap it to 24h or 48h ahead shown. That's plenty for live simulation – and no need to simulate beyond that; channels presumably loop or random beyond. If user tries to set a "reminder" or something beyond that, not needed in our scope.

Example: Suppose Channel 7 is "Comedy Shows" and has *The Office* and *Parks and Recreation*. We might decide channel cycles: 2 episodes of Office, then 2 episodes of Parks, and repeat. We randomly choose a starting point – say Parks was halfway through an episode at the moment channel is created. From that moment, we then alternate blocks. If a user tunes in at any time, they get whichever show's episodes should be on. After finishing all episodes of these series, we loop back to season 1 (or we could exclude ones watched? But since we aren't marking watched, we'll just loop endlessly, it's a channel). Over weeks, the user might eventually see all episodes of both series in order, then it repeats.

Seeking In-Progress Shows: When a user is watching a channel, we allow them to rewind a bit. Implementation: if they press rewind (the 10-second skip back, e.g.), we can just do `video.currentTime -= 10`. This doesn't change the channel schedule – from perspective of schedule, they are now "behind live". We can keep playing and the schedule will move on without them (conceptually). If they are behind and the show ends in real time, what do we do? We likely let them continue watching to the end of that program (even though the channel has technically moved on). Once they hit the program end (according to their delayed timeline), we could then jump them to live on the next program (or give an option: "Continue to live" or allow catching up by fast-forward). Because implementing a full buffer across program boundaries is complex (we'd have to buffer the next program for them even though we're not live – which we can if the file is local; actually we can because all content is available on demand, we could just seamlessly let them roll into the next item even though by the time they finish, the channel might be one show ahead or so). But from a UX perspective, it might be fine to always jump to live when a show finishes if they were behind. Perhaps better: If a user has paused or delayed, they likely know they're behind; we can present a "Go Live" button (like QuasiTV does [70]). If they don't press it, we might just keep them behind indefinitely – essentially they're now watching on their own time. But that breaks the concept of "live channel" somewhat. Real IPTV typically flushes you to live at some cutoff (like you can only rewind so far behind live). We might set a buffer limit, say 30 minutes. If you're more than 30 min behind, the app might warn and jump forward. We should define this clearly in user terms.

For simplicity: implement pause/rewind for current item only. If you reach the end of that item while behind, just jump to live at the next program. This balances user control and the linear nature.

Performance considerations: The scheduling engine will run on JS on the TV. It's mostly dealing with time arithmetic and array of schedule entries. 10 channels, each with, say, 50 items a day (assuming average 30 min episodes in 24h ~ 48 items), that's 500 entries – trivial to handle in memory. Even if user had 100 channels (unlikely and not UI-friendly to navigate), still fine. The engine's updates every minute or so is fine. We must avoid doing heavy operations on the main thread while video is playing – but a few math computations are fine. We'll ensure any larger operations (like initial library parse) happen either before playback or in small chunks (perhaps during an initial loading screen where we say "Generating channels..." for a few seconds, or lazy-generate channels one by one so the user can start watching one while others still finishing generation).

Extensibility: The engine is designed to accommodate different scheduling rules (shuffle vs ordered, blocks vs single episodes, inclusion/exclusion of certain content). We'll create data structures like:

```
channel = {
   id: 7,
   name: "Comedy Channel",
   contentPolicy: { type: 'multi-show', shows: [ {id:show1, blockSize:2},
{id:show2, blockSize:2} ], order: 'round-robin' },
   schedule: [ {start, end, mediaId, title, ...}, ... ],
   lastEpisodePerShow: { show1: episode5Id, show2: episode3Id }  // tracking
progress
}
```

When adding new content (say show2 gets a new episode), next time that show comes up, we'll include it if unwatched or if at end of list.

Validation: We'll test the scheduling by simulating a day and ensuring continuity. We should also check for any drift – e.g., if there are fractional seconds adding up (some episodes might have durations not multiple of second?), likely not an issue as durations from Plex are in milliseconds integers.

In conclusion, the Channel Scheduling Engine will ensure each channel is essentially a *never-ending playlist mapped onto time*. It abstracts the complexity from the rest of the app: the Player just asks "what do I play at this moment?" and the engine provides the item and start offset. This design allows "deterministic playback and seeking for in-progress shows" – by always calculating from the known timeline we avoid inconsistency. If two people had the same library and channel config, at the same wall-clock time they'd see the same thing on a given channel (one could even imagine sharing channel configs, though out of scope). The engine's determinism and timely updates are key to fulfilling the live TV illusion.

LG webOS Platform Considerations and Deployment

Developing on LG webOS entails adhering to certain guidelines and optimizing for the TV hardware. We address deployment modes, app store requirements, performance tuning specific to webOS, and user interaction specifics on LG TVs:

- **Packaged vs. Hosted App:** As discussed, we have the option to release as a packaged app (all code on device) or a hosted app (code fetched from a server). LG's own documentation notes that hosted apps can be updated anytime and are good if the app changes frequently, while packaged apps may perform slightly better and work offline [71] [72]. Considering our application:
- If distributed via the **LG Content Store**, it will likely need to be packaged (the store typically distributes packaged IPKs). However, LG does allow hosted apps in the store by essentially publishing the stub that points to your site; but then you must maintain hosting and ensure uptime.
- A hybrid approach is possible: use a packaged app but have it fetch certain data or even updated JS from our server if we want dynamic updates (though that can be tricky and might violate store policies if abused to change code).

We plan to **initially use Developer Mode (sideload)** for testing and perhaps a limited hosted deployment for beta (so we can push updates quickly while testing with users). For the consumer release, we aim for submission to the LG store as a packaged app. This means going through LG's approval process, which includes meeting their app criteria.

- **App Store Considerations:** LG's app approval will check for compliance with their guidelines (we should review the App Self Checklist [73]). Some relevant points likely:
- The app must not use unauthorized APIs or do anything that could destabilize the TV.
- It shouldn't contain hard-coded personal info or be easily crashed.
- Content-wise, since this app plays user-provided media, it should be fine. We have to ensure no piracy or something – but since Plex is the user's own server, it's user-controlled content. LG might require us to clarify that in the app description (as they might for Plex itself).
- If our app has any adult content considerations (it doesn't inherently, aside from what the user's library has), we might need a rating or parental control integration. Possibly we should hook into LG's parental control system if needed (likely not in scope, we assume if a user has Plex parental controls, they manage that on Plex side).

- We should provide the required assets: app icon, banner image, description, etc., for listing. And the app should handle various resolutions (1080p and 4K) gracefully. (We'll likely design at 1080p and let it scale on 4K).

- **Developer Mode and Testing:** During development, we'll frequently push the app to a TV in Dev Mode. We'll monitor logs using the `ares-inspect` or Dev Inspector to catch any errors. Particularly, memory usage will be watched – if we have memory leaks (e.g. not cleaning up DOM or not releasing video sources), the app could be killed by the system with an "out of memory" error (webOS will show a message "app needs to restart to free memory" if it grows too large [74]). We need to ensure that doesn't happen, especially with continuous use. Using LG's **Resource Monitor** tool [75] [76], we can profile memory and CPU while running our app to catch any problematic areas.

- **Performance Optimization on Low-End webOS Hardware:** Some entry-level LG TVs have slower processors. The UI should remain fluid at 60fps ideally, but if not possible, 30fps is acceptable for something like scrolling the guide. We should minimize heavy computations on the main thread. For example, rendering the EPG grid with too many elements can slow down – solution: virtualize the list of channels and times (render only what's visible). We can implement the guide such that it only creates DOM elements for maybe ~5 channels x 3 hours worth of programs that are on-screen, and reuse elements when scrolling (recycling with new data). Many TV app frameworks do this to handle large guides efficiently.

Also, as per smart TV performance guides, avoid large re-renders: e.g., don't constantly re-render the entire guide every second – instead update a small "current time" indicator. The blog we found emphasizes avoiding unnecessary re-render and using memoization [77], which we will do – e.g., in React, wrap components with `React.memo` so they only update when needed. The remote input has inherent latency (~100ms maybe), but we don't want to add to that with sluggish JS. By keeping state updates lean and using CSS transitions for focus changes (rather than recalculating layout in JS), we can achieve snappy navigation.

Garbage collection (GC) pauses can cause stutter. We'll try to avoid generating tons of throwaway objects every frame. The scheduling engine will reuse objects where possible, or use pools if needed (though likely not an issue at our scale).

The **memory limit** for web apps on LG can be around a few hundred MB before the system complains. We won't load huge assets, just images which might sum to tens of MB worst-case. Still, we should release references to old data (like if user regenerates channels, drop the old schedules, etc.). If we use images (posters), caching too many large ones could fill memory – we can rely on the browser cache for that and not keep all images in memory at full resolution (just let <img> elements pull them as needed).

- **Remote Control Nuances:** The LG Magic Remote has a pointer mode – in pointer mode, our app might receive mouse events instead of key events. We need to ensure things like clicking on a program in the guide with the pointer works (that likely means making them a clickable element). Also, when pointer is used, focus navigation might be suspended until the pointer is hidden or the user uses arrow keys (as doc says, moving with arrow will disable pointer mode [78]). We'll test both modes. We might also manually hide the pointer during video playback for a clean experience (the pointer usually disappears after a few seconds of no movement anyway).

- **Audio Focus:** If our app is playing audio (which it will), we should handle what happens if the user opens another app or hits the Home button – the audio should stop. The webOS system likely does this automatically (pauses our app or mutes it when not foreground). We might also implement onPause (lifecycle) to pause playback to be safe.

- **Miscellaneous webOS APIs:** Some webOS features that might be relevant:

- **Screensaver**: We don't want the TV's screensaver to activate while a channel is playing. Usually, video playback stops screensaver. But if a user pauses the video for a long time, screensaver might come on. That's fine.
- **Return key / Exit**: On older LG remotes there's an Exit key. Web apps might not get that (the system probably handles it by exiting the app). We'll assume Back long-press or Exit will close the app; we'll try to save state if that happens.

- **Notifications**: If needed, we could integrate with LG's notification system to show system banners for certain events, but probably not needed.

- **Analytics**: We might incorporate simple analytics (like tracking which features used) for development, but if we do, ensure user consents if required by policy (privacy guidelines apply, especially if any data leaves the device) [79].

- **Hosted App Deployment & Maintenance:** If we do decide to go the hosted route for distribution, we'd host the app's files on a CDN or server. We'd have to worry about **HTTPS** (required, and a proper certificate, as LG will not load insecure content). Also, any calls from the app to Plex (which could be local IP HTTP) might be mixed content if our app is loaded via HTTPS. This is tricky: e.g., if our hosted app is at `https://myapp.com`, and the Plex server is on `http://192.168.1.5:32400`, the browser might block that as mixed content (HTTPS page making HTTP request). Actually, that's likely. So, if hosted, we may need to use Plex's secure remote URL (which is HTTPS) even for local connections to avoid mixed content issues. Plex servers generate a self-signed cert and use Plex relay with plex.direct domain for secure connection. We can try to use those always. But direct local HTTP might be more efficient. On a packaged app, it's not an issue because it's not loaded via https (file:// or app:// context usually), so mixed content rules might not apply. We will have to test that. If mixed content is an issue, we'll default to using secure connections (which could slightly reduce local throughput but likely fine).

- **Continuous Delivery:** If we need to update the app often (maybe add features like recording or more integrations), the hosted model or a good update pipeline is needed. LG requires apps to be re-approved for each update in the store, which can take time. Using a hosted app means we can push updates instantly, but if our domain is down, the app fails. We'll weigh these trade-offs. Possibly, we could do as some do: package a minimal app that on launch fetches an index from our server (like a version check) and then loads either local code or updated code. But that starts to blur store policies.

In any event, for initial deployment we will likely target the LG Content Store. We will register as a developer with LG, create an app listing, and submit the app package. The app will undergo testing by LG for functionality (they'll test on various TV models to ensure no crashes, proper navigation, etc.). We'll ensure: - No debug overlays or logs are visible in release build. - The app handles network unavailability gracefully (e.g., if no internet and no Plex reachable, we show an error but app still responsive). - The app doesn't break the Home menu or other TV functions.

User Support and Maintenance: We might include a screen showing app version and perhaps logs to help debugging user issues. Also, as Plex or webOS updates in future, we'll need to maintain compatibility. For example, Plex might change their API (unlikely) or LG might update webOS with new web engine that might require slight adjustments (like a new user agent string detection etc.).

Future Extensions: Though not required in this spec, the architecture could allow adding a server-side component for more complex features (like remote management of channels, or syncing channels across devices). For now, we keep it all on the TV. Also, features like recording or pause-buffer persistence could be considered down the line (record would essentially be saving the currently playing segment to disk – but that's a DVR feature which might raise content rights issues, so probably not).

Privacy and Security: The app will handle the user's Plex credentials/token. We will follow privacy best practices: not share the token with anyone except Plex's servers, not log it, and provide a way to log out (Coax forgot a log out at first, as a Reddit user noted [80], and the dev added it). We'll include a log out that forgets the token and any cached data if the user desires. Our app does not send any data externally except to Plex (unless we implement analytics or error reporting, which if we do, will be anonymized and disclosed).

To sum up, deploying on webOS requires careful attention to remote control UX, performance tuning for limited hardware, and abiding by LG's app ecosystem rules. By leveraging LG's frameworks (Enact) and tools, and thoroughly testing on actual TVs, we aim to deliver an app that feels native to the platform and runs reliably 24/7 (if a user leaves it on a channel all day, it should continue without crashing or hogging memory). We will conduct soak tests (long-duration tests) to ensure memory doesn't creep up (simulating continuous playback for days). WebOS's resource monitor and logs will assist in this [76] [81]. The end result will be a well-integrated Plex "Virtual Channels" client that LG users can confidently install and enjoy as part of their TV experience.

User Interface Design & TV UX Best Practices

Designing for a 10-foot TV experience requires a different mindset than designing for mobile or desktop. In this section, we highlight the UI/UX choices that make the app intuitive and comfortable on LG TVs, incorporating general smart TV guidelines and specific behaviors observed in apps like QuasiTV and Coax:

- **Focus and Navigation:** On a TV, users navigate with directional keys, so every interactive element must be reachable by the D-pad. We will ensure a **logical focus order**: e.g., in the channel guide grid, pressing down moves to the same time on the next channel, pressing right moves to later time on the same channel, etc. We will implement wrap-around or boundary conditions thoughtfully (maybe pressing down on the last channel doesn't wrap to first unless we explicitly want circular navigation, which probably we don't for a long list; instead we might stop or pop a prompt "End of channel list"). Similarly, leftmost would either stop or if on guide maybe go to channel name column which is focusable as QuasiTV did [4] (they allowed focusing the channel name to then select the channel and go live [70]). We will highlight the focused cell or item with a distinct border or glow. The size of the focus should be large enough to see from far away. According to LG's style, they often use a light blue outline or a scale-up effect for focused buttons (Enact's default focus visual does a scale+glow). We'll utilize that so users always know where they are.

- **Large, Readable Text:** All text in the UI will use a sufficiently large font (e.g., 24px or larger for normal text, maybe 36px for titles). We will use a font that is legible on TV (likely LG's default font or a similar sans-serif). Content titles can be long, so in the EPG, if a title is too long to fit in the box, we might scroll it (marquee) or at least show full title in the description panel below. High contrast is key: a dark background and light text or vice versa. Our EPG design might mimic Plex's own guide or others, with dark translucent background and white text for unfocused, and maybe cyan highlight for focused. We'll test on an actual TV to ensure readability.

- **Colors and Safe Zones:** We'll stick to a limited color palette that suits TV. Possibly take inspiration from LG's **Sandstone** design language (which uses blues, greys, etc.). We will also heed the 20px safe

area margins [17] : our app's layout will have padding such that UI elements aren't at absolute pixel 0 or 1919 of a 1080p screen. Enact in fact automatically accounts for overscan if configured, but we will double-check by testing on a TV with overscan (some TVs in certain modes might cut a few pixels).

- **Orientation and Responsive UI:** All LG TVs are landscape 16:9. We don't need to support portrait or resizing. We will design mainly for 1080p. On 4K TVs, LG might either pixel-doubles 1080p apps or expects our app to render at 4K. Modern webOS might actually run the app at 4K resolution (there's an app resolution spec available [82] , possibly indicating we can declare if we use 4K assets). It's worth checking if we can/should mark our app as 4K capable, which allows sharper text on 4K screens. If so, we should supply higher-res graphics if any (but our UI is mostly drawn with CSS/text, so it will naturally be sharp). Regardless, our layout should remain consistent. Using relative sizes (em, vh) might help scaling across resolutions, but often we just target 1080p and trust the platform scaling for 4K. We'll verify on a 4K model for crispness.

- **Visual Feedback and Transitions:** Whenever a user does something (like pressing a button), the app should respond quickly with a visual cue. For example, if they press "Guide" (say we map the red button to open Guide), the guide should appear with a slight animation (maybe fade in). We will provide feedback for slow operations: if connecting to Plex is taking a couple seconds, show a loading spinner or progress ("Loading Plex content..."). Similarly, when switching a channel, if there's a buffering delay, we show a spinner or a "Loading..." text on a blank background, or perhaps keep showing last frame of previous channel until next is ready (though better to blank to avoid confusion). We should also handle invalid actions gracefully: e.g., if the user presses a key that does nothing in current context, either ignore it or play a subtle "bump" sound to indicate no action (sound is tricky since we don't have custom sounds typically in web apps; might skip sounds and rely on visuals).

- **Audio & Sound Effects:** We might integrate basic sound feedback using the web Audio API or playing a short sound file, but many TV apps keep it silent except content audio. Possibly using the LG **Multi-Sound** API [83] if needed, but likely overkill. The user's content provides the sound.

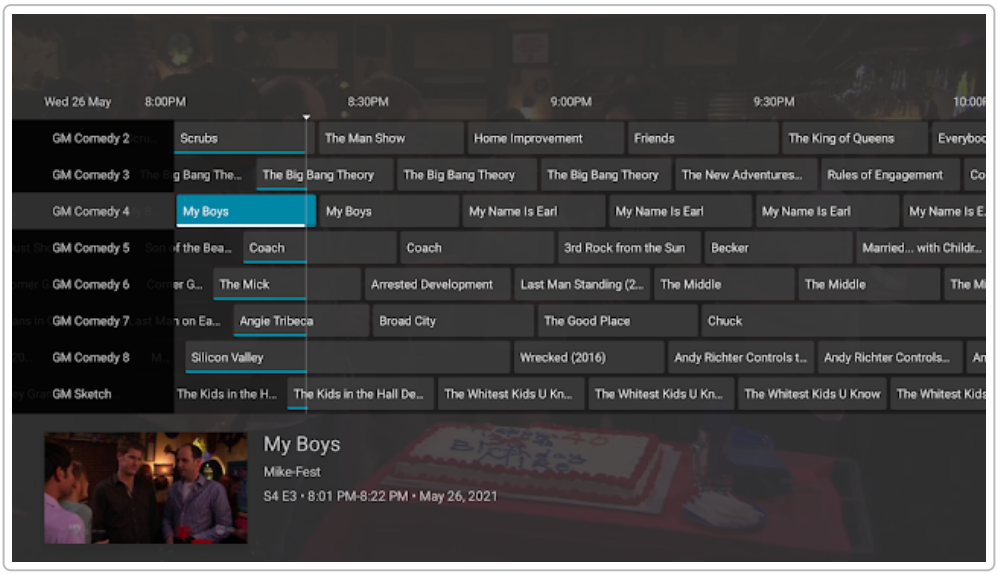
- **Guide Data Updating:** As time moves, the guide's highlight of current time should move. We could implement a vertical line across the EPG at current time that moves. Or simpler, just highlight the currently playing program in each row when guide is opened (QuasiTV 2.7 did that [4]). They highlight the program that is currently playing on that channel rather than always highlighting based on clock if the user has navigated. We will possibly do the same: when guide opens, snap focus to the current program on the current channel. If user scrolls around and then exits guide, fine.

- **Accessibility:** While TV apps typically don't have screen readers, LG does have voice guidance features. We are not focusing on that now, but using standard HTML elements and labels where possible could allow some integration. Enact also has accessibility support.

- **Testing UX with Users:** We might get some Plex power-users to try a beta and give feedback on channel generation fairness, UI ease of use, etc. We can incorporate such feedback especially because preferences vary (some might want a channel that's truly random, others want sequential but shuffled starting points, etc.). We can include toggles for shuffle vs sequential in channel settings, for example.

In the end, by adhering to established TV UI patterns and the specifics of LG's platform, the app should feel natural to users. The **learning curve** is minimal because the interactions mimic what users already know from TV: channel up/down, a guide grid, info banners, etc. We leverage muscle memory (e.g., "Press Up to see channel list" like QuasiTV did [84]). We'll document a few of these in an on-screen help or first-run tutorial (maybe a simple overlay highlighting "Press ▲ for Guide, ◀▶ to seek" etc.). Since remote controls vary (magic vs standard), we'll ensure those instructions make sense generally (e.g. say "Press OK to select a channel" rather than "click").

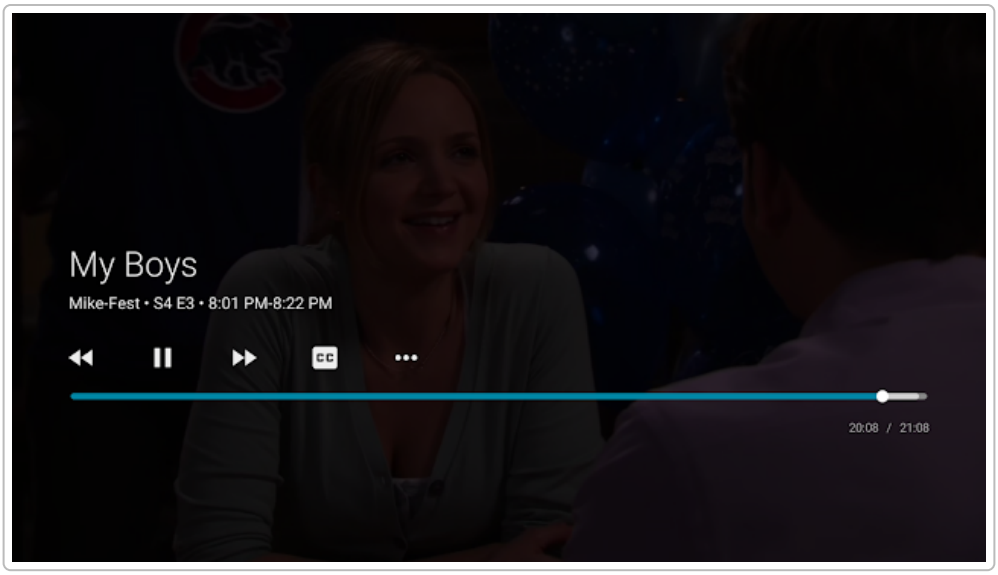
To illustrate, here's a mock of what the channel guide UI might look like in our app:



Example of an EPG (program guide) interface from QuasiTV's Android TV app, showing multiple custom channels in a timeline grid. Our app will implement a similar guide: channels listed vertically with their scheduled programs laid out horizontally, and a highlight on the current program with details shown below (title, episode info, etc.).

In our design, each channel might have a number or icon on the left, and program blocks colored or shaded. The current channel that user is/was watching could be emphasized. The bottom info area shows the program's title, description, and metadata when a program is focused. We'll replicate these patterns. Additionally, when just flipping channels without guide, a simple on-screen display (OSD) will appear – typically a semi-transparent bar with the channel name/number and current show info for a few seconds.

For the in-player UI (when pressing down or OK during playback), we plan a minimal control bar:



Example of a playback overlay from QuasiTV, showing on-screen controls and progress. Our app's playback UI will have similar elements: show title, episode info, a progress bar with current time/total time, and controls for play/pause, rewind 10s, fast-forward 10s, and CC toggle.

As shown above, pressing down (or OK) brings up controls: a play/pause button, rewind/forward icons (for skip), CC for subtitles, and a timeline bar [85] . The background dims or blurs. We will implement this control bar and hide it after a few seconds of no input. If the user presses left/right while the bar is up, we could treat that as seek (e.g., 10s increments) or navigate between buttons. We'll test what feels right. On some remotes, there are dedicated rewind/ff keys which we'll handle separately (they'll just perform the action directly without needing the UI up).

Remote Input Latency and Feedback: Because remote presses on a TV can feel laggy, we ensure our app responds immediately in UI even if an action (like loading channel) is still processing. For instance, if user presses channel down and we need 1 second to buffer video, we should immediately switch the channel label on screen to the new channel and show a spinner, so the user knows the command was received. Without feedback, they might press again and queue multiple commands. We'll also possibly implement a small sound or blink on selection to reassure input acceptance.

Hosted vs Packaged UI differences: If the app is hosted, initial load might show a blank or loading indicator – we'll add a branded splash screen (maybe the LG app splash that shows our logo and loading...). In packaged, the splash is basically the time it takes to init the app.

By focusing on these UI/UX details, we ensure that the app not only functions well but also delights the user with an authentic, comfortable "TV channels" experience. We marry Plex's rich media with the nostalgia and ease of channel surfing, all within the familiar confines of their LG TV interface.

Conclusion

In this implementation plan, we have outlined a comprehensive strategy to develop a scheduled playback "virtual TV channel" app on LG webOS, integrating deeply with Plex. We have defined how to replicate the linear channel experience provided by apps like Coax and QuasiTV, while tailoring it to webOS's technology stack and user expectations in 2025. From an architectural standpoint, the solution is a standalone webOS app that handles everything from user login to channel scheduling to media playback, using Plex as the content source.

Key highlights of the design include a deterministic channel scheduling engine (ensuring live TV behavior with join-in-progress and continuous play), an optimized HTML5 video playback strategy (leveraging webOS's native HLS and codec support for smooth direct-play and minimal transcoding), and a user interface that adheres to TV best practices (channel guide, remote-friendly navigation, big-screen readability). We have addressed Plex integration challenges like authentication via PIN linking [55] and efficient use of Plex APIs for metadata and streaming.

Performance and reliability are prioritized throughout: we'll use LG's Enact/React framework to build a robust UI while following performance best practices for smart TVs (avoiding unnecessary re-renders and heavy scripts to keep the app responsive [77]). We also incorporate LG-specific guidance for safe area display [17], remote control handling, and memory optimization. The app will be tested on real hardware to ensure that extended usage does not cause memory leaks or instability – critical for an app meant to run like a TV channel, potentially for hours on end.

Finally, we considered deployment strategies including using a hosted app model to allow iterative improvements and faster updates. If utilized, we will maintain a secure hosting and potentially update content on the fly, but ultimately we aim for a polished release through LG's store, complying with their ecosystem requirements. The collaboration of open-source insights (like QuasiTV's scheduling approach [9]) and official docs (like LG's media spec [40] and Plex's API guidelines) has informed this plan to ensure it is both **current (as of Dec 2025)** and grounded in proven techniques.

By implementing this plan, we will deliver to users a unique product: **their own personal "cable TV"** filled with content they love, running natively on their LG TV. It merges the old-school joy of channel surfing with modern streaming convenience, all while maintaining high performance and a seamless user experience. With a solid architecture and careful attention to technical details, we are well-positioned to successfully build and deploy the virtual channel app on webOS, bringing the Plex library to life in a new way.

Sources:

- LG webOS TV Developer documentation (APIs, media formats, and guidelines) – e.g., supported video formats and streaming protocol capabilities [45] [40], and UI/UX design guides [17].
- Community insights from QuasiTV (an Android/FireTV app with similar goals) – particularly on channel scheduling logic and Plex integration nuances [9] [3].
- Comments from the developer of Coax (beta Plex channel app) highlighting the conceptual model of only "playing what you're watching" with the rest being simulated [69].
- Smart TV performance optimization advice, emphasizing minimal re-renders and mindful state management for smooth app behavior on webOS's limited hardware [77] [86].

- Plex API usage guidelines and authentication flow instructions from Plex's own developers [55] [60], ensuring we follow Plex's best practices for login and API calls.

---

[1] [2] [47] [77] [86] Optimizing Performance in Smart TV Apps: A Developer's Guide | TO THE NEW Blog
https://www.tothenew.com/blog/optimizing-performance-in-smart-tv-apps-a-developers-guide/

[3] QuasiTV
https://www.quasitv.app/2021/03/

[4] [5] [10] [54] [70] QuasiTV 2.1.0 - Playlist / Collection Based Custom Channels
https://www.quasitv.app/2023/10/quasitv-210-playlist-collection-based.html

[6] [11] [12] [13] [14] [15] [16] [78] Magic Remote Control Unit | webOS TV Developer
https://webostv.developer.lge.com/develop/guides/magic-remote

[7] [69] [80] Coax beta open again! : r/PleX
https://www.reddit.com/r/PleX/comments/1pp8qyw/coax_beta_open_again/

[8] [19] Image Gallery
https://www.quasitv.app/p/image-gallery.html

[9] [27] [53] [64] [66] [84] QuasiTV
https://www.quasitv.app/2020/

[17] Overscan | webOS TV Developer
https://webostv.developer.lge.com/develop/guides/overscan

[18] Design Principal - webOS TV Developer - LG
https://webostv.developer.lge.com/develop/guides/design-principles

[20] [62] [63] [68] QuasiTV
https://www.quasitv.app/2022/03/

[21] [25] [26] [29] [79] [83] Enyo and Enact Guide | webOS TV Developer
https://webostv.developer.lge.com/develop/guides/enyo-enact-guide

[22] [23] [55] [56] [57] [58] [59] [60] [61] Authenticating with Plex - Dev/API Corner - Plex Forum
https://forums.plex.tv/t/authenticating-with-plex/609370

[24] [44] [45] [46] [49] [52] [65] [82] Audio and Video Format on webOS TV 6.0 | webOS TV Developer
https://webostv.developer.lge.com/develop/specifications/video-audio-60

[28] LG Electronics Enact | Showcase - Storybook
https://storybook.js.org/showcase/lg-electronics-enact

[30] [31] [32] [36] [37] [71] [72] [73] Web App Types | webOS TV Developer
https://webostv.developer.lge.com/develop/getting-started/web-app-types

[33] What is webOS? | Definition from TechTarget
https://www.techtarget.com/whatis/definition/webOS

[34] What is LG's webOS? All you need to know - Spyrosoft
https://spyro-soft.com/blog/media-and-entertainment/what-is-lgs-webos-all-you-need-to-know

35 webOS Studio Developer Guide
https://webostv.developer.lge.com/develop/tools/webos-studio-dev-guide

38 39 40 41 42 43 51 Streaming Protocol and DRM | webOS TV Developer
https://webostv.developer.lge.com/develop/specifications/streaming-protocol-drm

48 Web API and Web Engine | webOS TV Developer - LG
https://webostv.developer.lge.com/develop/specifications/web-api-and-web-engine

50 Plex | ErsatzTV
https://ersatztv.org/docs/clients/plex/

67 Scheduling question : r/QuasiTVAndroid - Reddit
https://www.reddit.com/r/QuasiTVAndroid/comments/1536n04/scheduling_question/

74 App keeps refreshing to free up memory. One day. My TV. LG ...
https://www.justanswer.com/tv-repair/sot3p-app-keeps-refreshing-free-memory-one-day-tv-lg.html

75 76 81 Resource Monitor Introduction | webOS TV Developer
https://webostv.developer.lge.com/develop/tools/resource-monitor-introduction

85 blogger.googleusercontent.com
https://blogger.googleusercontent.com/img/b/R29vZ2xl/
AVvXsEioG6Uzu1YMSd9pCJzIVl56RgUvQeyHh9NZBj547PGPLJRAOmZdrFFxMk17pHHk65V93uk0_BYWpmHoSnO9fxeiGsNBZI4SICl6YKpWsnn3sw1Gmq_H1
w640-h360/device-2021-05-26-202147.png