

TVS Hub

Source Code Recovery Guide

Date generated:	June 11, 2026
Author:	david2222221@gmail.com
Python:	3.12+
Framework:	PyQt6 + python-mpv + libmpv
Platform:	Linux (Ubuntu/Debian, X11)

This document contains every source file needed to recreate TVS Hub from scratch on a new machine. It includes step-by-step setup instructions followed by the complete source code for all modules.

1. System Requirements & Setup

System dependencies (install once with apt)

These packages must be installed before creating the Python virtual environment:

- python3.12
- python3.12-venv
- python3-pip
- libmpv-dev (or libmpv2)
- mpv (the command-line player)
- libx11-dev (X11 embedding support)

```
sudo apt install python3.12 python3.12-venv python3-pip libmpv-dev mpv libx11-dev
```

Project folder layout

```
LXPlayer/          <- project root
|-- main.py         <- entry point (run this)
|-- requirements.txt <- pip dependencies
`-- lxplayer/
    |-- __init__.py
    |-- app_logger.py <- singleton logger (Qt signal-based)
    |-- config.py    <- JSON config saved to ~/.config/tvshub/config.json
    |-- delegates.py <- custom row painter for the channel list
    |-- ep_g_screen.py <- Live-with-EPG (24h scrollable timeline)
    |-- home_screen.py <- home dashboard with gradient cards
    |-- log_panel.py  <- debug log dialog
    |-- login_dialog.py <- Xtream Codes credentials dialog
    |-- main_window.py <- main window + content browser (Live/Movies/Series)
    |-- multi_screen.py <- multi-screen picture-in-picture
    |-- player.py     <- MPV embedded player widget
    |-- utils.py      <- shared avatar / logo helpers
    |-- workers.py    <- QThreadPool background worker
    `-- xstream_api.py <- Xtream Codes HTTP API client
```

Creating the virtual environment & running the app

1. `mkdir ~/LXPlayer && cd ~/LXPlayer`
2. `# copy all source files into the folder (see Section 2)`
3. `python3 -m venv venv`
4. `source venv/bin/activate`
5. `pip install -r requirements.txt`
6. `python main.py`

Python dependencies (requirements.txt)

- PyQt6>=6.4.0
- python-mpv>=1.0.7
- requests>=2.28.0

Key features

- Live TV, Movies, Series browser backed by Xstream Codes API
- Live-with-EPG: scrollable 24-hour timeline with program info and channel logos
- Multi-Screen: up to 6 simultaneous panels, click any panel to go fullscreen
- Favorites & Recent virtual categories; right-click a channel to add/remove
- Recently Added virtual category in Movies tab — shows streams added within 30 days, sorted newest first
- Drag-to-reorder category list; order is saved per tab
- Channel logos fetched asynchronously; text-avatar fallback (initials + colour)
- Catch Up button opens the Favorites list instantly
- EPG now-playing subtitle shown under each channel name in the Live TV list
- Stream/series popup card: large poster, pill badges, synopsis (EPG for Live TV), full-width buttons
- Badge probing paused during video playback, continues during radio
- Stream failure cache: 800-day expiry. Season badge cache: 90-day expiry
- Keyboard shortcuts: Space = pause, F = fullscreen, Ctrl+L = re-login, Esc = back
- Configuration persisted to ~/.config/tvshub/config.json

Important notes for re-deployment

- The app requires X11 (not pure Wayland). XWayland works fine on most desktops.
- MPV is embedded via a native X11 window ID. PlayerWidget MUST be visible (in the live widget tree) before play() is called, or MPV will not initialise.
- MAX_EPG_CHANNELS = 500 in epg_screen.py to avoid X11's 32767 px height limit.
- The login dialog hard-codes server = http://tv10s.icu:8080 in credentials() (line 111 of login_dialog.py). Add a server field to the form if you need to connect to a different provider.
- Logo / icon fetching uses up to 4 concurrent workers per view to avoid overloading the provider server.
- Config is saved on window close and whenever favorites/category-order change.

2. Complete Source Code

install.sh

```
#!/usr/bin/env bash
# TVS Hub installer for Ubuntu / Debian-based Linux
# Extract TVSHub_Linux.tar.gz, then run from inside the tvshub/ folder:
#   bash install.sh

set -e

APP_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
INSTALL_DIR="$HOME/.local/share/tvshub"
BIN_LINK="$HOME/.local/bin/tvshub"
DESKTOP_FILE="$HOME/.local/share/applications/tvshub.desktop"
ICON_DIR="$HOME/.local/share/icons/hicolor/scalable/apps"

RED='\033[0;31m'; GREEN='\033[0;32m'; CYAN='\033[0;36m'
BOLD='\033[1m'; RESET='\033[0m'

info() { echo -e "${CYAN}${BOLD}[TVS Hub]${RESET} $*"; }
success() { echo -e "${GREEN}${BOLD}[✓]${RESET} $*"; }
die() { echo -e "${RED}${BOLD}[✗] ${RESET}" >&2; exit 1; }

# — sanity checks —
[[ "$EUID" -eq 0 ]] && die "Do not run as root – run as your normal user."
command -v apt-get &>/dev/null || die "This installer requires apt (Ubuntu / Debian)."
[[ -f "$APP_DIR/main.py" ]] || die "Run this script from inside the tvshub folder."

echo
echo -e "${BOLD} TVS Hub – Linux Installer${RESET}"
echo " _____"
echo

# — upgrade: remove old install if present —
if [[ -d "$INSTALL_DIR" ]]; then
    info "Existing installation found – removing before upgrade..."
    rm -rf "$INSTALL_DIR"
    rm -f "$BIN_LINK"
    rm -f "$DESKTOP_FILE"
    success "Old version removed. Your settings in ~/.config/tvshub/ are untouched."
    echo
fi

# — 1. system dependencies —
info "Installing system dependencies (requires sudo)..."
sudo apt-get update -qq
sudo apt-get install -y \
    python3 \
    python3-venv \
    python3-pip \
    libmpv-dev \
    mpv \
    ffmpeg \
    libx11-dev \
    fonts-dejavu-core
success "System packages installed."
```

```
# — 2. copy app files —————
info "Copying application files..."
mkdir -p "$INSTALL_DIR"
if command -v rsync &>/dev/null; then
    rsync -a --exclude='venv' --exclude='__pycache__' --exclude='*.pyc' \
        "$APP_DIR/" "$INSTALL_DIR/"
else
    cp -r "$APP_DIR"/. "$INSTALL_DIR/"
    find "$INSTALL_DIR" -name '__pycache__' -type d -exec rm -rf {} + 2>/dev/null || true
    find "$INSTALL_DIR" -name '*.pyc' -delete 2>/dev/null || true
fi
success "Files copied to $INSTALL_DIR"

# — 3. virtual environment —————
info "Setting up Python environment..."
python3 -m venv "$INSTALL_DIR/venv"
"$INSTALL_DIR/venv/bin/pip" install --upgrade pip -q
"$INSTALL_DIR/venv/bin/pip" install -r "$INSTALL_DIR/requirements.txt" -q
success "Python dependencies installed."

# — 4. launcher —————
info "Creating launcher..."
mkdir -p "$HOME/.local/bin"
cat > "$BIN_LINK" <<EOF
#!/usr/bin/env bash
export QT_QPA_PLATFORM=xcb
exec "$INSTALL_DIR/venv/bin/python" "$INSTALL_DIR/main.py" "$@"
EOF
chmod +x "$BIN_LINK"
success "Launcher created at $BIN_LINK"

# — 5. desktop shortcut —————
info "Creating desktop shortcut..."
mkdir -p "$HOME/.local/share/applications"
mkdir -p "$ICON_DIR"
cp "$INSTALL_DIR/assets/tvshub.svg" "$ICON_DIR/tvshub.svg"
gtk-update-icon-cache "$HOME/.local/share/icons/hicolor" 2>/dev/null || true
cat > "$DESKTOP_FILE" <<EOF
[Desktop Entry]
Version=1.0
Type=Application
Name=TVS Hub
Comment=IPTV player
Exec=$BIN_LINK
Icon=tvshub
Terminal=false
Categories=AudioVideo;Video;Player;
Keywords=iptv;tv;streams;tv;
StartupWMClass=TVS Hub
EOF
update-desktop-database "$HOME/.local/share/applications" 2>/dev/null || true
success "Desktop shortcut created."

# — 6. PATH check —————
if [[ ":$PATH:" != *"$HOME/.local/bin:*" ]]; then
    echo
    echo -e "${CYAN}Note:${RESET} ~/.local/bin is not in your PATH."
    echo "    To use the 'tvshub' command, run:"
    echo "        echo 'export PATH=\"$HOME/.local/bin:$PATH\"' >> ~/.bashrc"
```

```
    echo    "    source ~/.bashrc"
fi

echo
echo -e "${GREEN}${BOLD}TVS Hub installed successfully!${RESET}"
echo    "    Run from terminal : tvshub"
echo    "    Or open 'TVS Hub' from your application menu."
echo
```

uninstall.sh

```
#!/usr/bin/env bash
# TVS Hub uninstaller
# Run from anywhere:  bash uninstall.sh

set -e

INSTALL_DIR="$HOME/.local/share/tvshub"
BIN_LINK="$HOME/.local/bin/tvshub"
DESKTOP_FILE="$HOME/.local/share/applications/tvshub.desktop"

RED='\033[0;31m'; GREEN='\033[0;32m'; CYAN='\033[0;36m'
BOLD='\033[1m'; RESET='\033[0m'

info()    { echo -e "${CYAN}${BOLD}[TVS Hub]${RESET} $*"; }
success() { echo -e "${GREEN}${BOLD}[✓]${RESET} $*"; }

echo
echo -e "${BOLD}  TVS Hub – Uninstaller${RESET}"
echo    "  _____"
echo

# confirm
read -rp "  Remove TVS Hub and all its files? [y/N] " confirm
[[ "$confirm" =~ ^[Yy]$ ]] || { echo "Cancelled."; exit 0; }
echo

info "Removing application files..."
rm -rf "$INSTALL_DIR"
success "Removed $INSTALL_DIR"

info "Removing launcher..."
rm -f "$BIN_LINK"
success "Removed $BIN_LINK"

info "Removing desktop shortcut..."
rm -f "$DESKTOP_FILE"
update-desktop-database "$HOME/.local/share/applications" 2>/dev/null || true
success "Removed $DESKTOP_FILE"

echo
echo -e "${GREEN}${BOLD}TVS Hub has been uninstalled.${RESET}"
echo    "  Your config and favourites are kept at:"
echo    "  ~/.config/tvshub/config.json"
echo    "  Delete that file too if you want a completely clean removal."
echo
```

assets/tvshub.svg

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 128 128" width="128" height="128">
  <defs>
    <linearGradient id="bodyGrad" x1="0" y1="0" x2="0" y2="1">
      <stop offset="0%" stop-color="#1b3a5c"/>
      <stop offset="100%" stop-color="#0d1b2a"/>
    </linearGradient>
    <linearGradient id="screenGrad" x1="0" y1="0" x2="1" y2="1">
      <stop offset="0%" stop-color="#1565c0"/>
      <stop offset="50%" stop-color="#42a5f5"/>
      <stop offset="100%" stop-color="#00e5ff"/>
    </linearGradient>
    <linearGradient id="legGrad" x1="0" y1="0" x2="0" y2="1">
      <stop offset="0%" stop-color="#1b3a5c"/>
      <stop offset="100%" stop-color="#0d1b2a"/>
    </linearGradient>
  </defs>

  <!-- legs -->
  <rect x="38" y="98" width="10" height="18" rx="3" fill="url(#legGrad)"/>
  <rect x="80" y="98" width="10" height="18" rx="3" fill="url(#legGrad)"/>
  <!-- leg feet -->
  <rect x="33" y="113" width="20" height="5" rx="2.5" fill="#0d1b2a"/>
  <rect x="75" y="113" width="20" height="5" rx="2.5" fill="#0d1b2a"/>

  <!-- TV body -->
  <rect x="8" y="22" width="112" height="78" rx="10" fill="url(#bodyGrad)"/>
  <!-- body highlight rim -->
  <rect x="8" y="22" width="112" height="78" rx="10"
    fill="none" stroke="#1976d2" stroke-width="1.5" opacity="0.6"/>

  <!-- screen bezel -->
  <rect x="16" y="30" width="82" height="58" rx="6" fill="#050e1a"/>
  <!-- screen -->
  <rect x="19" y="33" width="76" height="52" rx="4" fill="url(#screenGrad)" opacity="0.92"/>
  <!-- screen glare -->
  <rect x="21" y="35" width="30" height="10" rx="3" fill="white" opacity="0.08"/>

  <!-- TVS Hub text on screen -->
  <text x="57" y="63" font-family="Arial,sans-serif" font-size="11"
    font-weight="bold" fill="white" text-anchor="middle"
    letter-spacing="1">TVS Hub</text>

  <!-- right-side controls panel -->
  <circle cx="111" cy="48" r="4" fill="#1976d2" opacity="0.9"/>
  <circle cx="111" cy="61" r="2.5" fill="#0d47a1" opacity="0.8"/>
  <circle cx="111" cy="71" r="2.5" fill="#0d47a1" opacity="0.8"/>

  <!-- antenna left -->
  <line x1="44" y1="22" x2="30" y2="6" stroke="#1976d2" stroke-width="3"
    stroke-linecap="round"/>
  <circle cx="30" cy="6" r="3" fill="#42a5f5"/>
  <!-- antenna right -->
  <line x1="64" y1="22" x2="78" y2="6" stroke="#1976d2" stroke-width="3"
    stroke-linecap="round"/>
  <circle cx="78" cy="6" r="3" fill="#42a5f5"/>
</svg>
```

main.py

```
import sys
import os
import faulthandler
faulthandler.enable(all_threads=True)

# Force PyInstaller to bundle the email package by importing it explicitly.
# urllib3 (used by requests) imports email at runtime and PyInstaller
# won't include it unless it's referenced in the entry point.
import email
import email.mime
import email.mime.text
import email.mime.multipart
import email.mime.base
import email.message
import email.parser
import email.header
import email.utils
import email.generator
import email.charset
import email.encoders
import email.feedparser
import email.errors
import email.policy

# When frozen by PyInstaller, add the bundle directory to PATH so
# libmpv-2.dll (bundled alongside the exe) can be found by python-mpv.
if getattr(sys, "frozen", False):
    _bundle = sys._MEIPASS # type: ignore[attr-defined]
    os.environ["PATH"] = _bundle + os.pathsep + os.environ.get("PATH", "")

def main():
    # On Linux, force X11 for MPV window embedding.
    # If you're on a pure Wayland setup without XWayland, comment this out
    # and set QT_QPA_PLATFORM=wayland - embedded video won't work there.
    if sys.platform.startswith("linux"):
        os.environ.setdefault("QT_QPA_PLATFORM", "xcb")

    from PyQt6.QtWidgets import QApplication
    app = QApplication(sys.argv)
    app.setApplicationName("TVS Hub")
    app.setStyle("Fusion")

    from lxplayer.config import Config
    from lxplayer.main_window import MainWindow

    config = Config()
    window = MainWindow(config)
    window.show()

    sys.exit(app.exec())

if __name__ == "__main__":
    main()
```


requirements.txt

```
PyQt6>=6.4.0
python-mpv>=1.0.7
requests>=2.28.0
```

lxplayer/__init__.py**lxplayer/config.py**

```
import json
import os
import sys
import time
from pathlib import Path
from dataclasses import dataclass, asdict, field

if sys.platform == "win32":
    CONFIG_DIR = Path(os.environ.get("APPDATA", Path.home())) / "tvshub"
else:
    CONFIG_DIR = Path.home() / ".config" / "tvshub"
CONFIG_FILE = CONFIG_DIR / "config.json"
CONNECTION_FILE = CONFIG_DIR / "connection.json"
QUALITY_CACHE_FILE = CONFIG_DIR / "quality_cache.json"
VOD_QUALITY_CACHE_FILE = CONFIG_DIR / "vod_quality_cache.json"
SEASON_CACHE_FILE = CONFIG_DIR / "season_cache.json"
POSTER_CACHE_DIR = CONFIG_DIR / "posters"

@dataclass
class Account:
    username: str = ""
    password: str = ""
    label: str = "" # friendly name shown in saved-accounts list

@dataclass
class Config:
    account: Account = field(default_factory=Account)
    accounts: list = field(default_factory=list) # saved provider accounts (max 5)
    volume: int = 80
    recent_streams: list = field(default_factory=list)
    favorite_streams: list = field(default_factory=list)
    cat_order: dict = field(default_factory=dict)
    watch_progress: dict = field(default_factory=dict) # "vod:sid" / "ep:eid" -> {pos, name, ts}
    cc_enabled: bool = False
    player_prefs: dict = field(default_factory=dict) # content_type -> player_id
    extra_players: list = field(default_factory=list) # [(path, name), ...]
    buffer_secs: int = 10 # MPV read-ahead buffer (seconds)
    radio_stream_ids: list = field(default_factory=list) # stream_ids learned as audio-only
    radio_favorite_ids: list = field(default_factory=list) # stream_ids starred as radio favourites
    radio_favorite_streams: list = field(default_factory=list) # full stream dicts for radio favourites
    radio_recent_streams: list = field(default_factory=list) # recently played radio stations (most recent first, max > 20)
    hidden_stream_ids: list = field(default_factory=list) # stream_ids hidden by the user
```

```
hidden_category_ids: list = field(default_factory=list) # category_ids hidden by the user
working_radio_ids: list = field(default_factory=list) # stream_ids verified as working radio
hide_channel_prefixes: bool = True # strip country codes from display names
stream_failures: dict = field(default_factory=dict) # {stream_id: {count, last_fail}}

def __post_init__(self):
    self._load()

def _acc(self, d: dict) -> Account:
    return Account(
        username=d.get("username", ""),
        password=d.get("password", ""),
        label=d.get("label", ""),
    )

def _load(self):
    # credentials
    if CONNECTION_FILE.exists():
        try:
            data = json.loads(CONNECTION_FILE.read_text())
            acc = data.get("account", {})
            self.account = self._acc(acc) if acc else Account()
            self.accounts = [self._acc(a) for a in data.get("accounts", [])]
        except Exception:
            pass
    elif CONFIG_FILE.exists():
        # migrate credentials from old single-file config
        try:
            data = json.loads(CONFIG_FILE.read_text())
            acc = data.get("account", {})
            self.account = self._acc(acc) if acc else Account()
            self.accounts = [self._acc(a) for a in data.get("accounts", [])]
        except Exception:
            pass

    # main settings
    if not CONFIG_FILE.exists():
        self._load_default_backup()
        return
    try:
        data = json.loads(CONFIG_FILE.read_text())
        self.volume = data.get("volume", 80)
        self.recent_streams = data.get("recent_streams", [])
        self.favorite_streams = data.get("favorite_streams", [])
        self.cat_order = data.get("cat_order", {})
        self.watch_progress = data.get("watch_progress", {})
        self.cc_enabled = bool(data.get("cc_enabled", False))
        self.player_prefs = data.get("player_prefs", {})
        self.extra_players = data.get("extra_players", [])
        self.buffer_secs = int(data.get("buffer_secs", 10))
        self.radio_stream_ids = data.get("radio_stream_ids", [])
        self.radio_favorite_ids = data.get("radio_favorite_ids", [])
        self.radio_favorite_streams = data.get("radio_favorite_streams", [])
        self.radio_recent_streams = data.get("radio_recent_streams", [])
        self.hidden_stream_ids = data.get("hidden_stream_ids", [])
        self.hidden_category_ids = data.get("hidden_category_ids", [])
        self.working_radio_ids = data.get("working_radio_ids", [])
        self.hide_channel_prefixes = bool(data.get("hide_channel_prefixes", True))
        self.stream_failures = data.get("stream_failures", {})
        self._clean_stream_failures()
```

```
except Exception:
    pass

def _load_default_backup(self):
    candidates = [
        Path(sys.executable).parent / "tvshub_backup.json",
        Path(__file__).parent.parent / "tvshub_backup.json",
    ]
    if getattr(sys, "frozen", False):
        import sys as _sys
        candidates.insert(0, Path(_sys._MEIPASS) / "tvshub_backup.json")
    for path in candidates:
        if path.exists():
            try:
                data = json.loads(path.read_text())
                if not data.get("tvshub_backup"):
                    continue
                for field in ("favorite_streams", "cat_order", "hidden_stream_ids",
                             "hidden_category_ids", "radio_favorite_ids",
                             "radio_favorite_streams"):
                    if field in data:
                        setattr(self, field, data[field])
            except Exception:
                pass
            break

def save(self):
    CONFIG_DIR.mkdir(parents=True, exist_ok=True)
    CONNECTION_FILE.write_text(json.dumps({
        "account": asdict(self.account),
        "accounts": [asdict(a) for a in self.accounts],
    }, indent=2))
    CONFIG_FILE.write_text(json.dumps({
        "volume": self.volume,
        "recent_streams": self.recent_streams,
        "favorite_streams": self.favorite_streams,
        "cat_order": self.cat_order,
        "watch_progress": self.watch_progress,
        "cc_enabled": self.cc_enabled,
        "player_prefs": self.player_prefs,
        "extra_players": self.extra_players,
        "buffer_secs": self.buffer_secs,
        "radio_stream_ids": self.radio_stream_ids,
        "radio_favorite_ids": self.radio_favorite_ids,
        "radio_favorite_streams": self.radio_favorite_streams,
        "radio_recent_streams": self.radio_recent_streams,
        "hidden_stream_ids": self.hidden_stream_ids,
        "hidden_category_ids": self.hidden_category_ids,
        "working_radio_ids": self.working_radio_ids,
        "hide_channel_prefixes": self.hide_channel_prefixes,
        "stream_failures": self.stream_failures,
    }, indent=2))

def _clean_stream_failures(self):
    cutoff = time.time() - 800 * 86400
    self.stream_failures = {
        sid: e for sid, e in self.stream_failures.items()
        if e.get("last_fail", 0) > cutoff
    }
```

```

def record_stream_failure(self, stream_id: str) -> None:
    if not stream_id:
        return
    e = self.stream_failures.get(stream_id, {"count": 0, "last_fail": 0})
    e["count"] = e["count"] + 1
    e["last_fail"] = int(time.time())
    self.stream_failures[stream_id] = e
    self.save()

def clear_stream_failure(self, stream_id: str) -> None:
    if stream_id in self.stream_failures:
        del self.stream_failures[stream_id]
        self.save()

def is_stream_unreliable(self, stream_id: str) -> bool:
    e = self.stream_failures.get(stream_id)
    if not e or e.get("count", 0) < 1:
        return False
    return (time.time() - e.get("last_fail", 0)) < 800 * 86400

def save_account(self, account: "Account"):
    """Upsert account into saved list (max 5, most-recent first)."""
    self.accounts = [
        a for a in self.accounts
        if a.username != account.username
    ]
    self.accounts.insert(0, account)
    self.accounts = self.accounts[:5]
    self.save()

@property
def has_account(self) -> bool:
    return bool(self.account.username)

```

lxplayer/app_logger.py

```

from PyQt6.QtCore import QObject, pyqtSignal
from datetime import datetime

class AppLogger(QObject):
    entry_added = pyqtSignal(str, str) # (level, message)

    _instance = None

    @classmethod
    def get(cls) -> "AppLogger":
        if cls._instance is None:
            cls._instance = cls()
        return cls._instance

    def _emit(self, level: str, msg: str):
        ts = datetime.now().strftime("%H:%M:%S")
        line = f"[{ts}] {msg}"
        print(line)
        self.entry_added.emit(level, line)

    def info(self, msg: str):

```

```
self._emit("INFO", msg)

def error(self, msg: str):
    self._emit("ERROR", msg)

def request(self, url: str, params: dict):
    self._emit("REQ", f"GET {url}\n          params: {params}")

def response(self, status: int, body: str):
    preview = body[:400] + ("..." if len(body) > 400 else "")
    self._emit("RESP", f"HTTP {status} {preview}")
```

lxplayer/workers.py

```
from PyQt6.QtCore import QRunnable, QObject, pyqtSignal

class WorkerSignals(QObject):
    result = pyqtSignal(object)
    error = pyqtSignal(str)

class Worker(QRunnable):
    def __init__(self, fn, *args, **kwargs):
        super().__init__()
        self.fn = fn
        self.args = args
        self.kwargs = kwargs
        self.signals = WorkerSignals()
        self.setAutoDelete(True)

    def run(self):
        try:
            self.signals.result.emit(self.fn(*self.args, **self.kwargs))
        except Exception as exc:
            self.signals.error.emit(str(exc))
```

lxplayer/xtream_api.py

```
import sys
import threading
import requests
from typing import Optional
from requests.utils import quote
from .app_logger import AppLogger

if sys.platform == "win32":
    _USER_AGENT = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 Chrome/120.0.0.0 Safari/537.36"
else:
    _USER_AGENT = "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 Chrome/120.0.0.0 Safari/537.36"
_local = threading.local()

def _get_session() -> requests.Session:
    """Return a per-thread Session so concurrent workers don't share state."""
    if not hasattr(_local, "session"):
```

```
s = requests.Session()
s.headers.update({"User-Agent": _USER_AGENT})
_local.session = s
return _local.session

class XtreamAPI:
    def __init__(self, server: str, username: str, password: str):
        self.server = server.rstrip("/")
        self.username = username
        self.password = password

    @property
    def _auth(self) -> dict:
        return {"username": self.username, "password": self.password}

    def _get(self, **params):
        log = AppLogger.get()
        full_params = {**self._auth, **params}
        url = f"{self.server}/player_api.php"
        safe_params = {k: ("***" if k in ("password", "username") else v)
                       for k, v in full_params.items()}
        log.request("player_api.php", safe_params)
        try:
            resp = _get_session().get(url, params=full_params, timeout=20)
            log.response(resp.status_code, resp.text)
            resp.raise_for_status()
            return resp.json()
        except Exception as exc:
            log.error(str(exc))
            raise

    def authenticate(self) -> dict:
        return self._get()

    def get_live_categories(self) -> list:
        return self._get(action="get_live_categories")

    def get_live_streams(self, category_id: Optional[str] = None) -> list:
        params = {"action": "get_live_streams"}
        if category_id:
            params["category_id"] = category_id
        return self._get(**params)

    def get_vod_categories(self) -> list:
        return self._get(action="get_vod_categories")

    def get_vod_streams(self, category_id: Optional[str] = None) -> list:
        params = {"action": "get_vod_streams"}
        if category_id:
            params["category_id"] = category_id
        return self._get(**params)

    def get_vod_info(self, vod_id) -> dict:
        return self._get(action="get_vod_info", vod_id=vod_id)

    def get_live_info(self, stream_id) -> dict:
        return self._get(action="get_live_info", stream_id=stream_id)

    def get_series_categories(self) -> list:
```

```

        return self._get(action="get_series_categories")

    def get_series(self, category_id: Optional[str] = None) -> list:
        params = {"action": "get_series"}
        if category_id:
            params["category_id"] = category_id
        return self._get(**params)

    def get_short_epg(self, stream_id, limit: int = 2) -> dict:
        return self._get(action="get_short_epg", stream_id=stream_id, limit=limit)

    def _u(self) -> str:
        return quote(self.username, safe="")

    def _p(self) -> str:
        return quote(self.password, safe="")

    def live_url(self, stream_id) -> str:
        return f"{self.server}/live/{self._u()}/{self._p()}/{stream_id}.m3u8"

    def movie_url(self, stream_id, ext: str = "mp4") -> str:
        return f"{self.server}/movie/{self._u()}/{self._p()}/{stream_id}.{ext}"

    def get_series_info(self, series_id) -> dict:
        return self._get(action="get_series_info", series_id=series_id)

    def series_episode_url(self, episode_id, ext: str = "mp4") -> str:
        return f"{self.server}/series/{self._u()}/{self._p()}/{episode_id}.{ext}"

```

lxplayer/utils.py

```

import re
import requests as _requests

_RADIO_CAT_KEYWORDS = {"radio", "fm", "am", "wireless"}
_RADIO_NAME_RE = re.compile(
    r'\b(radio)\b'          # "Radio" as a standalone word
    r'|\bfm\b'             # FM as a standalone word (011.FM, Hit FM, 1 Faith FM, etc.)
    r'|\bfm\d'             # FM followed by digits (e.g. FM4)
    r'|\d+[.,]\d+s*fm\b'   # frequency notation (98.5 FM)
    r'|\b\d+s*fm\b'        # integer frequency (104 FM)
    r'|\bam\s+radio\b',    # "AM Radio"
    re.IGNORECASE,
)

def is_radio_stream(s: dict) -> bool:
    """Return True if the stream dict represents a radio station."""
    if s.get("is_radio") in (1, "1", True):
        return True
    cat = (s.get("category_name") or "").lower()
    if any(kw in cat for kw in _RADIO_CAT_KEYWORDS):
        return True
    name = s.get("name") or ""
    return bool(_RADIO_NAME_RE.search(name))

from PyQt6.QtCore import Qt
from PyQt6.QtGui import QColor, QFont, QPainter, QPainterPath, QPixmap

```

```
_AVATAR_COLORS = [
    "#1565c0", "#00695c", "#6a1b9a", "#c62828",
    "#e65100", "#558b2f", "#37474f", "#4a148c",
    "#1976d2", "#00897b", "#7b1fa2", "#d32f2f",
    "#f57c00", "#689f38", "#455a64", "#512da8",
]

def avatar_initials(name: str) -> str:
    name = re.sub(r'^[A-Z]{2,3}:\s*', '', name).strip()
    words = [w for w in name.split() if w.upper() not in ("HD", "THE", "A", "AN")]
    if not words:
        return name[:1].upper() or "?"
    if len(words) == 1:
        return words[0][:2].upper()
    return (words[0][0] + words[1][0]).upper()

def avatar_color(name: str) -> str:
    return _AVATAR_COLORS[sum(ord(c) for c in name) % len(_AVATAR_COLORS)]

def fetch_logo_bytes(url: str, timeout: int = 4) -> bytes:
    if not url:
        return b""
    try:
        resp = _requests.get(url, headers={"User-Agent": "Mozilla/5.0"}, timeout=timeout)
        resp.raise_for_status()
        return resp.content
    except Exception:
        return b""

def round_pixmap(px: QPixmap, size: int, radius: int = 7) -> QPixmap:
    scaled = px.scaled(size, size,
                       Qt.AspectRatioMode.KeepAspectRatio,
                       Qt.TransformationMode.SmoothTransformation)
    out = QPixmap(size, size)
    out.fill(Qt.GlobalColor.transparent)
    p = QPainter(out)
    p.setRenderHint(QPainter.RenderHint.Antialiasing)
    path = QPainterPath()
    path.addRoundedRect(0.0, 0.0, float(size), float(size), float(radius), float(radius))
    p.setClipPath(path)
    ox = (size - scaled.width()) // 2
    oy = (size - scaled.height()) // 2
    p.drawPixmap(ox, oy, scaled)
    p.end()
    return out

def make_avatar(name: str, size: int) -> QPixmap:
    px = QPixmap(size, size)
    px.fill(Qt.GlobalColor.transparent)
    p = QPainter(px)
    p.setRenderHint(QPainter.RenderHint.Antialiasing)
    p.setRenderHint(QPainter.RenderHint.TextAntialiasing)
    path = QPainterPath()
    path.addRoundedRect(1.0, 1.0, size - 2.0, size - 2.0, 7.0, 7.0)
```



```

    p.fillPath(path, QColor(avatar_color(name)))
    initials = avatar_initials(name)
    f = QFont()
    f.setPixelSize(size // 2 if len(initials) == 1 else size // 3)
    f.setBold(True)
    p.setFont(f)
    p.setPen(QColor("white"))
    p.drawText(px.rect(), Qt.AlignmentFlag.AlignCenter, initials)
    p.end()
    return px

def make_radio_avatar(size: int) -> QPixmap:
    """Fallback avatar for radio stations – fixed dark background with a emoji."""
    px = QPixmap(size, size)
    px.fill(Qt.GlobalColor.transparent)
    p = QPainter(px)
    p.setRenderHint(QPainter.RenderHint.Antialiasing)
    p.setRenderHint(QPainter.RenderHint.TextAntialiasing)
    path = QPainterPath()
    path.addRoundedRect(1.0, 1.0, size - 2.0, size - 2.0, 7.0, 7.0)
    p.fillPath(path, QColor("#1a3a4a"))
    f = QFont()
    f.setPixelSize(int(size * 0.55))
    p.setFont(f)
    p.drawText(px.rect(), Qt.AlignmentFlag.AlignCenter, "")
    p.end()
    return px

```

lxplayer/player.py

```

import sys
from PyQt6.QtWidgets import QWidget, QLabel, QVBoxLayout
from PyQt6.QtCore import Qt, pyqtSignal, QTimer

class PlayerWidget(QWidget):
    clicked = pyqtSignal()
    double_clicked = pyqtSignal()
    audio_only_changed = pyqtSignal(bool) # True = no video track detected
    resolution_changed = pyqtSignal(str) # "1080p", "720p", "SD", etc.

    def __init__(self, parent=None):
        super().__init__(parent)
        self.setAttribute(Qt.WidgetAttribute.WA_DontCreateNativeAncestors)
        self.setAttribute(Qt.WidgetAttribute.WA_NativeWindow)
        # Don't take keyboard focus – keeps app shortcuts working while playing.
        self.setFocusPolicy(Qt.FocusPolicy.NoFocus)
        self.setMinimumSize(640, 360)
        self.setStyleSheet("background: black;")
        self._mpv = None
        self._error: str = ""

        # Loading / patience overlay. MPV renders into this widget's native
        # window, so the overlay must itself be a native window raised on top
        # (a plain Qt child would be drawn under the video). Solid background
        # so the text is always readable across platforms.
        self._msg = QLabel("", self)

```

```
self._msg.setWordWrap(True)
self._msg.setAlignment(Qt.AlignmentFlag.AlignCenter)
self._msg.setAttribute(Qt.WidgetAttribute.WA_NativeWindow)
self._msg.setStyleSheet(
    "background:#000000; color:#cfe0f5; font-size:16px; padding:24px;"
)
self._msg.hide()

# MPV's video window is created (and can be re-created) slightly after
# play() is called, and it lands on top of the overlay. Re-raise the
# overlay for a short window so the message stays visible until the
# stream actually starts and hide_message() is called.
self._raise_timer = QTimer(self)
self._raise_timer.setInterval(120)
self._raise_timer.timeout.connect(self._reraise_msg)
self._raise_ticks = 0

# External players (e.g. embedded VLC) emit no MPV signals, so the
# overlay is dismissed by this fallback timer instead.
self._autohide_timer = QTimer(self)
self._autohide_timer.setSingleShot(True)
self._autohide_timer.timeout.connect(self.hide_message)

def show_message(self, text: str, auto_hide_ms: int = 0):
    self._msg.setText(text)
    self._msg.setGeometry(0, 0, self.width(), self.height())
    self._msg.show()
    self._msg.raise_()
    self._raise_ticks = 0
    self._raise_timer.start()
    self._autohide_timer.stop()
    if auto_hide_ms > 0:
        self._autohide_timer.start(auto_hide_ms)

def _reraise_msg(self):
    self._raise_ticks += 1
    if self._msg.isVisible():
        self._msg.setGeometry(0, 0, self.width(), self.height())
        self._msg.raise_()
    # keep re-raising for ~6s, then give up (stream may just be slow)
    if self._raise_ticks >= 50 or not self._msg.isVisible():
        self._raise_timer.stop()

def hide_message(self):
    self._raise_timer.stop()
    self._autohide_timer.stop()
    self._msg.hide()

def resizeEvent(self, event):
    if self._msg.isVisible():
        self._msg.setGeometry(0, 0, self.width(), self.height())
    super().resizeEvent(event)

def _init_mpv(self, buffer_secs: int = 10):
    from .app_logger import AppLogger
    log = AppLogger.get()
    wid = int(self.winId())
    log.info(f"MPV init wid={wid} size={self.width()}x{self.height()}")
    try:
        import mpv
```

```
def _mpv_log(level, component, message):
    if message and message.strip():
        AppLogger.get().info(f"[mpv/{component}] {message.strip()}")

mb = max(16, buffer_secs * 2)
if sys.platform == "win32":
    ua = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 Chrome/120.0.0.0 Safari/537.36"
else:
    ua = "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 Chrome/120.0.0.0 Safari/537.36"
kwargs = dict(
    wid=str(wid),
    keep_open=True,
    idle=True,
    hwdec="auto",
    log_handler=_mpv_log,
    loglevel="warn",
    user_agent=ua,
    demuxer_readahead_secs=buffer_secs,
    demuxer_max_bytes=f"{mb}MiB",
    cache=True,
    network_timeout=15,
    # Don't let MPV grab the keyboard – the app owns shortcuts.
    input_default_bindings=False,
    input_vo_keyboard=False,
)
if sys.platform == "win32":
    kwargs["vo"] = "gpu"          # Windows: DirectX/OpenGL, no X11
elif sys.platform != "darwin":
    kwargs["vo"] = "gpu,x11"     # Linux: fall back to x11 if gpu fails
self._mpv = mpv.MPV(**kwargs)

# Mouse clicks are handled exclusively by Qt's mousePressEvent below,
# so we do not register MPV key handlers for MBTN_LEFT etc.
# Doing so caused the clicked signal to fire multiple times per click
# (once from MPV's thread, once from Qt), making fullscreen toggle
# enter then immediately exit.

@self._mpv.property_observer('width')
def _on_width(name, value):
    if value:
        h = self._mpv.height or 0
        self.resolution_changed.emit(self._res_label(value, h))

@self._mpv.event_callback('file-loaded')
def _on_file_loaded():
    tracks = self._mpv.track_list or []
    has_video = any(t.get("type") == "video" for t in tracks)
    self.audio_only_changed.emit(not has_video)
    if not has_video:
        self.resolution_changed.emit("")

log.info("MPV initialized OK")
except ImportError:
    self._error = "python-mpv not found. Install with: pip install python-mpv"
    log.error(self._error)
    self._show_error()
except Exception as exc:
    self._error = str(exc)
    log.error(f"MPV init failed: {exc}")
```

```
        self._show_error()

def _show_error(self):
    lbl = QLabel(f"Player unavailable:\n{self._error}")
    lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
    lbl.setStyleSheet("color: #e57373; font-size: 12px; padding: 16px;")
    lbl.setWordWrap(True)
    layout = QVBoxLayout(self)
    layout.addWidget(lbl)

def mousePressEvent(self, event):
    if event.button() == Qt.MouseButton.LeftButton:
        import time
        from .app_logger import AppLogger
        AppLogger.get().info(f"[PLR] click t={time.monotonic():.3f}")
        self.clicked.emit()
        event.accept()
        return
    super().mousePressEvent(event)

def mouseDoubleClickEvent(self, event):
    if event.button() == Qt.MouseButton.LeftButton:
        self.double_clicked.emit()
        event.accept()
        return
    super().mouseDoubleClickEvent(event)

def play(self, url: str, buffer_secs: int = 10, is_live: bool = False):
    if self._mpv is None and not self._error:
        self._init_mpv(buffer_secs)
    if self._mpv:
        if is_live:
            # Live / radio: short readahead, no freeze-stall on low buffer.
            self._mpv.cache_pause = False
            self._mpv.demuxer_readahead_secs = min(buffer_secs, 5)
            self._mpv.demuxer_max_bytes = "10MiB"
        else:
            # VOD: buffer generously for smooth playback.
            self._mpv.cache_pause = True
            self._mpv.demuxer_readahead_secs = max(buffer_secs, 20)
            self._mpv.demuxer_max_bytes = f"{max(40, buffer_secs * 2)}MiB"
        self._mpv.play(url)

def stop(self):
    if self._mpv:
        self._mpv.stop()

def toggle_pause(self):
    if self._mpv:
        self._mpv.pause = not self._mpv.pause

@property
def paused(self) -> bool:
    return bool(self._mpv.pause) if self._mpv else True

def set_volume(self, value: int):
    if self._mpv:
        self._mpv.volume = max(0, min(130, value))

@property
```

```
def volume(self) -> int:
    return int(self._mpv.volume or 80) if self._mpv else 80

def toggle_fullscreen(self):
    if self._mpv:
        self._mpv.fullscreen = not self._mpv.fullscreen

# — track selection —————

def audio_tracks(self) -> list:
    if not self._mpv:
        return []
    try:
        return [t for t in (self._mpv.track_list or []) if t.get("type") == "audio"]
    except Exception:
        return []

def sub_tracks(self) -> list:
    if not self._mpv:
        return []
    try:
        return [t for t in (self._mpv.track_list or []) if t.get("type") == "sub"]
    except Exception:
        return []

def set_audio_track(self, tid):
    if self._mpv:
        try:
            self._mpv.aid = tid
        except Exception:
            pass

def set_sub_track(self, tid):
    if self._mpv:
        try:
            self._mpv.sid = "no" if tid == 0 else tid
        except Exception:
            pass

@property
def sub_enabled(self) -> bool:
    if not self._mpv:
        return False
    try:
        return self._mpv.sid not in (None, "no", 0)
    except Exception:
        return False

# — playback position —————

@property
def duration(self) -> float:
    if self._mpv:
        try:
            v = self._mpv.duration
            return float(v) if v is not None else 0.0
        except Exception:
            return 0.0
    return 0.0
```

```
@property
def time_pos(self) -> float:
    if self._mpv:
        try:
            v = self._mpv.time_pos
            return float(v) if v is not None else 0.0
        except Exception:
            return 0.0
    return 0.0

@property
def paused_for_cache(self) -> bool:
    if self._mpv:
        try:
            return bool(self._mpv.paused_for_cache or False)
        except Exception:
            return False
    return False

@property
def cache_pct(self) -> int:
    if self._mpv:
        try:
            v = self._mpv.cache_buffering_state
            return int(v) if v is not None else 0
        except Exception:
            return 0
    return 0

@property
def cache_secs(self) -> int:
    """Seconds of media buffered ahead of the playhead (demuxer cache)."""
    if self._mpv:
        try:
            v = self._mpv.demuxer_cache_duration
            return int(v) if v is not None else 0
        except Exception:
            return 0
    return 0

def seek_to(self, seconds: float):
    if self._mpv:
        try:
            self._mpv.seek(seconds, "absolute")
        except Exception:
            pass

    @staticmethod
    def _res_label(w: int, h: int) -> str:
        if h >= 2160 or w >= 3840: return "4K"
        if h >= 1080 or w >= 1920: return "1080p"
        if h >= 720 or w >= 1280: return "720p"
        if h >= 480 or w >= 854: return "480p"
        return "SD"

# — stream diagnostics —————

def stream_info(self) -> dict:
    if not self._mpv:
        return {}
```

```
info = {}
try:
    w, h = self._mpv.width, self._mpv.height
    if w and h:
        info["Resolution"] = f"{w}x{h}"
    vc = self._mpv.video_codec
    if vc:
        info["Video"] = vc
    ac = self._mpv.audio_codec
    if ac:
        info["Audio"] = ac
    fps = self._mpv.estimated_vf_fps
    if fps:
        info["FPS"] = f"{fps:.1f}"
except Exception:
    pass
return info

@property
def error(self) -> str:
    return self._error

def terminate_mpv(self):
    """Fully terminate MPV so its X11 window handle is free for another process."""
    if self._mpv:
        try:
            self._mpv.terminate()
        except Exception:
            pass
        self._mpv = None

def closeEvent(self, event):
    self.terminate_mpv()
    super().closeEvent(event)
```

lxplayer/player_select_screen.py

```
import shutil
import sys
import os
import socket as _socket
import threading

from PyQt6.QtWidgets import (
    QWidget, QVBoxLayout, QHBoxLayout, QLabel, QPushButton,
    QFileDialog, QMenu, QComboBox, QCheckBox,
)
from PyQt6.QtCore import Qt, pyqtSignal

# — content types shown in the table —————
CONTENT_TYPES = [
    ("live", "LIVE TV"),
    ("movies", "MOVIES"),
    ("series", "SERIES"),
    ("radio", "RADIO"),
    ("multiscreen", "MULTI-SCREEN"),
]
```

```
BUILTIN_ID = "builtin"
BUILTIN_NAME = "Built-in Player (Hardware / Software Decoder)"

_WINDOWS_PLAYER_PATHS = [
    (r"C:\Program Files\VideoLAN\VLC\vlc.exe", "VLC Media Player"),
    (r"C:\Program Files (x86)\VideoLAN\VLC\vlc.exe", "VLC Media Player"),
    (r"C:\Program Files\mpv\mpv.exe", "MPV Player (standalone window)"),
    (r"C:\Program Files\MPC-HC\mpc-hc64.exe", "MPC-HC"),
    (r"C:\Program Files\MPC-HC\mpc-hc.exe", "MPC-HC"),
]

def _detect_players() -> list[tuple[str, str]]:
    """Return (path, label) for media players found on this machine."""
    if sys.platform == "win32":
        seen_labels = set()
        found = []
        for path, label in _WINDOWS_PLAYER_PATHS:
            if os.path.isfile(path) and label not in seen_labels:
                found.append((path, label))
                seen_labels.add(label)
        return found
    else:
        candidates = [
            ("vlc", "VLC Media Player"),
            ("mpv", "MPV Player (standalone window)"),
            ("celluloid", "Celluloid (GNOME MPV)"),
            ("smplayer", "SMPlayer"),
            ("mplayer", "MPlayer"),
            ("totem", "Totem"),
            ("dragon", "Dragon Player"),
        ]
        return [(cmd, label) for cmd, label in candidates if shutil.which(cmd)]

def _windows_default_player() -> str | None:
    """Return path of the first detected player on Windows."""
    if sys.platform != "win32":
        return None
    players = _detect_players()
    return players[0][0] if players else None

def _size_args(player_id: str, w: int, h: int) -> list[str]:
    name = player_id.split("\\")[-1].split("/")[1].lower()
    if "vlc" in name:
        return [f"--width={w}", f"--height={h}"]
    if "mpv" in name:
        return [f"--geometry={w}x{h}"]
    if "smplayer" in name:
        return []
    if "mplayer" in name:
        return ["-xy", str(w)]
    return []

def launch_external(player_id: str, url: str, extra_players: list,
                    width: int = 0, height: int = 0, wid: int = 0,
                    vlc_rc_port: int = 0):
    """Launch url in an external player. Returns the Popen handle, or None."""
```



```
import subprocess
if player_id in ("builtin", ""):
    return None
cmd = None
if sys.platform == "win32":
    if os.path.isfile(player_id):
        cmd = [player_id]
else:
    if shutil.which(player_id):
        cmd = [player_id]
if cmd is None:
    for path, _ in extra_players:
        if path == player_id:
            cmd = [path]
            break
if cmd is None:
    return None

name = cmd[0].split("\\")[-1].split("/")[1].lower()
vlc_embedded = "vlc" in name and wid > 0

if vlc_embedded:
    if sys.platform == "win32":
        cmd += [
            f"--drawable-hwnd={wid}",
            "--no-video-title-show",
            "--no-osd",
            "--intf", "dummy",
            "--quiet",
            "--verbose=-1",
        ]
        if vlc_rc_port:
            cmd += ["--extraintf", "rc",
                    "--rc-host", f"127.0.0.1:{vlc_rc_port}",
                    "--rc-quiet"]
    else:
        cmd += [
            f"--drawable-xid={wid}",
            "--no-video-title-show",
            "--no-osd",
            "--no-mouse-events", # prevent VLC from handling double-click fullscreen
            "--intf", "dummy",
            "--extraintf", "rc",
            "--rc-fake-tty",
        ]
elif width > 0 and height > 0:
    cmd += _size_args(cmd[0], width, height)

cmd.append(url)
try:
    if vlc_embedded and sys.platform != "win32":
        # RC interface over stdio so we can poll playback status.
        return subprocess.Popen(cmd, stdin=subprocess.PIPE,
                                stdout=subprocess.PIPE, stderr=subprocess.DEVNULL)
    if sys.platform == "win32":
        # Suppress the console window that Windows shows for every subprocess.
        return subprocess.Popen(cmd,
                                stdin=subprocess.DEVNULL,
                                stdout=subprocess.DEVNULL,
                                stderr=subprocess.DEVNULL,
```

```
        creationflags=subprocess.CREATE_NO_WINDOW)

    return subprocess.Popen(cmd)
except Exception:
    return None

class VlcRcMonitor:
    """Polls an embedded VLC's RC interface (over stdio) for playback time.

    Only the playback clock is available from VLC – enough to drive the
    stream-health dot (advancing = alive). VLC exposes no buffer-cushion
    metric, so there is no seconds readout for it.
    """

    def __init__(self, proc):
        self._proc = proc
        self._time = 0
        self._lock = threading.Lock()
        self._running = True
        threading.Thread(target=self._read, daemon=True).start()

    def _read(self):
        out = getattr(self._proc, "stdout", None)
        if out is None:
            return
        while self._running:
            try:
                line = out.readline()
            except Exception:
                break
            if not line:
                break
            s = line.strip()
            while s.startswith(b">"):
                s = s[1:].strip()
            if s.isdigit():
                with self._lock:
                    self._time = int(s)

    def request(self):
        """Ask VLC for the current time (response is read by the thread)."""
        try:
            if self._proc.stdin and self._proc.poll() is None:
                self._proc.stdin.write(b"get_time\n")
                self._proc.stdin.flush()
        except Exception:
            pass

    @property
    def time(self) -> int:
        with self._lock:
            return self._time

    def stop(self):
        self._running = False

def _find_free_port() -> int:
    """Return an available local TCP port."""
    with _socket.socket() as s:
```

```
s.bind(('127.0.0.1', 0))
return s.getsockname()[1]
```

```
class VlcTcpMonitor:
    """Polls embedded VLC's RC interface over a local TCP socket (Windows).

    VLC must be launched with --extraintf rc --rc-host 127.0.0.1:PORT.
    Connects once VLC is ready and polls get_time every 500 ms.
    """

    def __init__(self, port: int):
        self._port = port
        self._time = 0
        self._lock = threading.Lock()
        self._running = True
        threading.Thread(target=self._run, daemon=True).start()

    def _run(self):
        import socket, time
        sock = None
        for _ in range(20):          # up to 10 s for VLC to start
            if not self._running:
                return
            try:
                s = socket.socket()
                s.settimeout(2.0)
                s.connect(('127.0.0.1', self._port))
                sock = s
                break
            except Exception:
                time.sleep(0.5)
        if sock is None:
            return
        buf = b""
        while self._running:
            try:
                sock.sendall(b"get_time\n")
                chunk = sock.recv(256)
                if not chunk:
                    break
                buf += chunk
                while b"\n" in buf:
                    line, buf = buf.split(b"\n", 1)
                    token = line.strip()
                    while token.startswith(b">"):
                        token = token[1:].strip()
                    if token.isdigit():
                        with self._lock:
                            self._time = int(token)
                    time.sleep(0.5)
            except Exception:
                break
            try:
                sock.close()
            except Exception:
                pass

    def request(self):
        pass # TCP thread polls on its own schedule
```

```

@property
def time(self) -> int:
    with self._lock:
        return self._time

def stop(self):
    self._running = False

# — per-row player selector button —————

class _SelectorBtn(QPushButton):
    def __init__(self, ct_id: str, prefs: dict, extra_players: list, parent=None):
        super().__init__(parent)
        self._ct_id = ct_id
        self._prefs = prefs
        self._extra = extra_players
        self.setStyleSheet(
            "QPushButton{background:#142030;color:#c0c8d8;border:1px solid #2a3a4a;"
            "border-radius:4px;font-size:12px;text-align:left;padding:0 12px;}"
            "QPushButton:hover{background:#1a2a3c;color:white;border-color:#3a5a7a;}"
            "QPushButton:focus{border:2px solid #42a5f5;outline:none;}"
        )
        self._refresh()
        self.clicked.connect(self._show_menu)

    def _all_players(self) -> list[tuple[str, str]]:
        players = [(BUILTIN_ID, BUILTIN_NAME)] + _detect_players()
        for path, name in self._extra:
            players.append((path, name))
        return players

    def _refresh(self):
        pid = self._prefs.get(self._ct_id, BUILTIN_ID)
        for p_id, p_name in self._all_players():
            if p_id == pid:
                self.setText(f" {p_name} ▼")
                return
        self.setText(f" {BUILTIN_NAME} ▼")

    def _show_menu(self):
        menu = QMenu(self)
        menu.setStyleSheet(
            "QMenu{background:#0f1e2d;color:#c0c8d8;border:1px solid #2a3a4a;"
            "border-radius:6px;padding:4px 0;}"
            "QMenu::item{padding:10px 24px;font-size:12px;}"
            "QMenu::item:selected{background:#1565c0;color:white;}"
            "QMenu::separator{height:1px;background:#1a2a3a;margin:4px 0;}"
        )
        cur = self._prefs.get(self._ct_id, BUILTIN_ID)
        for p_id, p_name in self._all_players():
            act = menu.addAction(p_name)
            act.setData(p_id)
            if p_id == cur:
                act.setCheckable(True)
                act.setChecked(True)
        chosen = menu.exec(self.mapToGlobal(self.rect().bottomLeft()))
        if chosen and chosen.data():
            self._prefs[self._ct_id] = chosen.data()

```

```
        self._refresh()

# — main screen —————

_BUFFER_OPTIONS = [
    (10, "Default (10s)"),
    (30, "Medium (30s)"),
    (60, "Large (60s)"),
    (120, "Extra Large (120s)"),
]

class PlayerSelectScreen(QWidget):
    back_clicked = pyqtSignal()

    def __init__(self, config, parent=None):
        super().__init__(parent)
        self._config = config
        self._prefs = dict(getattr(config, "player_prefs", {}))
        self._extra = list(getattr(config, "extra_players", []))
        self._buffer_secs = int(getattr(config, "buffer_secs", 10))
        self._hide_prefixes = bool(getattr(config, "hide_channel_prefixes", True))

        self.setStyleSheet("background:#0d1b2a;")
        self._row_btns: dict[str, _SelectorBtn] = {}
        self._build_ui()

# — public —————

    def open(self):
        self._prefs = dict(getattr(self._config, "player_prefs", {}))
        self._extra = list(getattr(self._config, "extra_players", []))
        self._buffer_secs = int(getattr(self._config, "buffer_secs", 10))
        self._hide_prefixes = bool(getattr(self._config, "hide_channel_prefixes", True))
        # sync combo box
        for i, (secs, _) in enumerate(_BUFFER_OPTIONS):
            if secs == self._buffer_secs:
                self._buffer_combo.setCurrentIndex(i)
                break
        self._prefix_chk.setChecked(self._hide_prefixes)
        for ct_id, btn in self._row_btns.items():
            btn._prefs = self._prefs
            btn._extra = self._extra
            btn._refresh()

# — internals —————

    def _build_ui(self):
        vl = QVBoxLayout(self)
        vl.setContentsMargins(0, 0, 0, 0)
        vl.setSpacing(0)

        hdr = QWidget()
        hdr.setFixedHeight(56)
        hdr.setStyleSheet("background:#0b1522;border-bottom:1px solid #1a2a3a;")
        hl = QHBoxLayout(hdr)
        hl.setContentsMargins(18, 0, 18, 0)
        hl.setSpacing(10)
```

```
back_arrow = QPushButton("-")
back_arrow.setFixedSize(34, 34)
back_arrow.setStyleSheet(
    "QPushButton{background:transparent;color:#42a5f5;border:none;font-size:20px;}"
    "QPushButton:hover{color:white;}"
    "QPushButton:focus{border:2px solid #42a5f5;border-radius:4px;outline:none;}"
)
back_arrow.clicked.connect(self._save_and_back)
hl.addWidget(back_arrow)

breadcrumb = QLabel("Settings | Player Selection")
breadcrumb.setStyleSheet("color:#7a9cc0;font-size:13px;background:transparent;")
hl.addWidget(breadcrumb)
hl.addStretch()

add_btn = QPushButton(" + ADD PLAYER")
add_btn.setStyleSheet(
    "QPushButton{background:#0f2035;color:#42a5f5;border:1px solid #1e4060;"
    "border-radius:6px;font-size:12px;font-weight:bold;padding:7px 18px;}"
    "QPushButton:hover{background:#142840;color:white;}"
    "QPushButton:focus{border:2px solid #42a5f5;outline:none;}"
)
add_btn.clicked.connect(self._add_player)
hl.addWidget(add_btn)
vl.addWidget(hdr)

title_lbl = QLabel("SELECT YOUR MEDIA PLAYER")
title_lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
title_lbl.setStyleSheet(
    "color:white;font-size:20px;font-weight:bold;"
    "padding:32px 0 28px 0;background:transparent;"
)
vl.addWidget(title_lbl)

rows_w = QWidget()
rows_w.setStyleSheet("background:transparent;")
rows_vl = QVBoxLayout(rows_w)
rows_vl.setContentsMargins(80, 0, 80, 0)
rows_vl.setSpacing(10)

for ct_id, ct_label in CONTENT_TYPES:
    row = QWidget()
    row.setFixedHeight(58)
    row.setStyleSheet(
        "QWidget{background:#0f1e2d;border:1px solid #1a3050;"
        "border-radius:6px;}"
    )
    rl = QHBoxLayout(row)
    rl.setContentsMargins(24, 0, 16, 0)
    rl.setSpacing(16)

    lbl = QLabel(ct_label)
    lbl.setStyleSheet(
        "color:white;font-size:14px;font-weight:bold;"
        "background:transparent;border:none;"
    )
    rl.addWidget(lbl, stretch=1)

    btn = _SelectorBtn(ct_id, self._prefs, self._extra, self)
    btn.setFixedSize(360, 38)
```

```
        self._row_btns[ct_id] = btn
        rl.addWidget(btn)

    rows_vl.addWidget(row)

vl.addWidget(rows_w)

# — buffer size row —————
buf_w = QWidget()
buf_w.setStyleSheet("background:transparent;")
buf_hl = QHBoxLayout(buf_w)
buf_hl.setContentsMargins(80, 12, 80, 4)
buf_hl.setSpacing(16)

buf_row = QWidget()
buf_row.setFixedHeight(58)
buf_row.setStyleSheet(
    "QWidget{background:#0f1e2d;border:1px solid #1a3050;border-radius:6px;}"
)
buf_rl = QHBoxLayout(buf_row)
buf_rl.setContentsMargins(24, 0, 16, 0)
buf_rl.setSpacing(16)

buf_lbl = QLabel("BUFFER SIZE")
buf_lbl.setStyleSheet(
    "color:white;font-size:14px;font-weight:bold;background:transparent;border:none;"
)
buf_rl.addWidget(buf_lbl, stretch=1)

self._buffer_combo = QComboBox()
self._buffer_combo.setFixedSize(260, 38)
self._buffer_combo.setStyleSheet(
    "QComboBox{background:#142030;color:#c0c8d8;border:1px solid #2a3a4a;"
    "border-radius:4px;font-size:12px;padding:0 12px;}"
    "QComboBox:hover{background:#1a2a3c;color:white;border-color:#3a5a7a;}"
    "QComboBox:focus{border:2px solid #42a5f5;outline:none;}"
    "QComboBox::drop-down{border:none;width:24px;}"
    "QComboBox QAbstractItemView{background:#0f1e2d;color:#c0c8d8;"
    "border:1px solid #2a3a4a;selection-background-color:#1565c0;}"
)
cur = int(getattr(self._config, "buffer_secs", 10))
for secs, label in _BUFFER_OPTIONS:
    self._buffer_combo.addItem(label, secs)
    if secs == cur:
        self._buffer_combo.setCurrentIndex(self._buffer_combo.count() - 1)
buf_rl.addWidget(self._buffer_combo)

buf_hl.addWidget(buf_row)
vl.addWidget(buf_w)

# — hide channel prefixes row —————
pfx_w = QWidget()
pfx_w.setStyleSheet("background:transparent;")
pfx_hl = QHBoxLayout(pfx_w)
pfx_hl.setContentsMargins(80, 0, 80, 4)
pfx_hl.setSpacing(16)

pfx_row = QWidget()
pfx_row.setFixedHeight(58)
pfx_row.setStyleSheet(
```

```
"QWidget{background:#0f1e2d;border:1px solid #1a3050;border-radius:6px;}"
)
pfx_rl = QHBoxLayout(pfx_row)
pfx_rl.setContentsMargins(24, 0, 16, 0)
pfx_rl.setSpacing(16)

pfx_lbl = QLabel("HIDE CHANNEL PREFIXES")
pfx_lbl.setStyleSheet(
    "color:white;font-size:14px;font-weight:bold;background:transparent;border:none;"
)
pfx_rl.addWidget(pfx_lbl, stretch=1)

self._prefix_chk = QCheckBox()
self._prefix_chk.setChecked(self._hide_prefixes)
self._prefix_chk.setStyleSheet(
    "QCheckBox{background:transparent;border:none;}"
    "QCheckBox::indicator{width:22px;height:22px;border:2px solid #2a3a4a;"
    "border-radius:4px;background:#142030;}"
    "QCheckBox::indicator:checked{background:#1565c0;border-color:#42a5f5;}"
    "QCheckBox::indicator:hover{border-color:#42a5f5;}"
    "QCheckBox:focus{outline:none;}"
)
pfx_rl.addWidget(self._prefix_chk)

pfx_hl.addWidget(pfx_row)
vl.addWidget(pfx_w)
vl.addStretch()

bot = QHBoxLayout()
bot.setContentsMargins(80, 16, 80, 32)
bot.setSpacing(16)

reset_btn = QPushButton("RESET")
reset_btn.setFixedHeight(46)
reset_btn.setStyleSheet(
    "QPushButton{background:qlineargradient(x1:0,y1:0,x2:1,y2:0,"
    "stop:0 #00c49a,stop:1 #1976d2);color:white;border:none;"
    "border-radius:6px;font-size:14px;font-weight:bold;}"
    "QPushButton:hover{background:qlineargradient(x1:0,y1:0,x2:1,y2:0,"
    "stop:0 #00d4a8,stop:1 #1e88e5);}"
    "QPushButton:focus{border:2px solid #42a5f5;outline:none;}"
)
reset_btn.clicked.connect(self._reset)
bot.addWidget(reset_btn, stretch=1)

back_btn2 = QPushButton("SAVE & CONTINUE")
back_btn2.setFixedHeight(46)
back_btn2.setStyleSheet(
    "QPushButton{background:#1a2a3a;color:white;border:none;"
    "border-radius:6px;font-size:14px;font-weight:bold;}"
    "QPushButton:hover{background:#243547;}"
    "QPushButton:focus{border:2px solid #42a5f5;outline:none;}"
)
back_btn2.clicked.connect(self._save_and_back)
bot.addWidget(back_btn2, stretch=1)

vl.addLayout(bot)

def _add_player(self):
    start = r"C:\Program Files" if sys.platform == "win32" else "/usr/bin"
```



```

    filt = "Executables (*.exe)" if sys.platform == "win32" else "All Files (*)"
    path, _ = QFileDialog.getOpenFileName(
        self, "Select Player Executable", start, filt
    )
    if not path:
        return
    name = path.split("\\")[-1].split("/")[-1].capitalize()
    entry = (path, name)
    if entry not in self._extra:
        self._extra.append(entry)
    for btn in self._row_btns.values():
        btn._extra = self._extra
        btn._refresh()

def _reset(self):
    self._prefs.clear()
    for btn in self._row_btns.values():
        btn._refresh()
    self._buffer_combo.setCurrentIndex(0)
    self._prefix_chk.setChecked(True)

def _save_and_back(self):
    self._config.player_prefs = dict(self._prefs)
    self._config.extra_players = list(self._extra)
    self._config.buffer_secs = _BUFFER_OPTIONS[self._buffer_combo.currentIndex()][0]
    self._config.hide_channel_prefixes = self._prefix_chk.isChecked()
    self._config.save()
    self.back_clicked.emit()

```

lxplayer/delegates.py

```

from datetime import datetime

from PyQt6.QtWidgets import QStyledItemDelegate, QStyle
from PyQt6.QtCore import Qt, QSize, QRect, QPoint
from PyQt6.QtGui import QPainter, QColor, QFont, QPen, QIcon

from .utils import make_avatar, make_radio_avatar

EPG_ROLE = Qt.ItemDataRole.UserRole + 1
EPG_PROG_ROLE = Qt.ItemDataRole.UserRole + 2
RADIO_ROLE = Qt.ItemDataRole.UserRole + 3
FAV_ROLE = Qt.ItemDataRole.UserRole + 4
QUALITY_ROLE = Qt.ItemDataRole.UserRole + 5
FAIL_ROLE = Qt.ItemDataRole.UserRole + 7

_ICON_SIZE = 36

_STAR_SIZE = 20 # px - size of the star hit/draw area
_STAR_PAD = 4 # px - distance from top-left corner of poster

class PosterDelegate(QStyledItemDelegate):
    """Draws a movie/series poster with a star overlay in the top-left corner."""

    def __init__(self, poster_w: int, poster_h: int, parent=None):
        super().__init__(parent)
        self.poster_w = poster_w

```

```
self.poster_h = poster_h

def _icon_rect(self, cell: QRect) -> QRect:
    """Return the poster rect centred within the grid cell."""
    x = cell.left() + (cell.width() - self.poster_w) // 2
    y = cell.top() + (cell.height() - self.poster_h) // 2
    return QRect(x, y, self.poster_w, self.poster_h)

def star_rect(self, cell: QRect) -> QRect:
    """Return the star hit area for a given cell rect (call from outside too)."""
    ir = self._icon_rect(cell)
    return QRect(ir.left() + _STAR_PAD, ir.top() + _STAR_PAD, _STAR_SIZE, _STAR_SIZE)

def paint(self, painter: QPainter, option, index):
    painter.save()

    # — placeholder (no stream data) —————
    if index.data(Qt.ItemDataRole.UserRole) is None:
        text = index.data(Qt.ItemDataRole.DisplayRole) or ""
        if text:
            painter.setRenderHint(QPainter.RenderHint.TextAntialiasing)
            painter.setPen(QColor("#4a6a8a"))
            f = QFont(); f.setPixelSize(14)
            painter.setFont(f)
            painter.drawText(option.rect, Qt.AlignmentFlag.AlignCenter, text)
        painter.restore()
        return

    painter.setRenderHint(QPainter.RenderHint.Antialiasing)

    r = option.rect
    selected = bool(option.state & QStyle.StateFlag.State_Selected)

    # — selection highlight —————
    if selected:
        painter.fillRect(r, QColor("#0d2044cc"))
        painter.setPen(QPen(QColor("#42a5f5"), 2))
        painter.drawRect(r.adjusted(1, 1, -1, -1))
    else:
        painter.fillRect(r, QColor(0, 0, 0, 0))

    # — poster image —————
    ir = self._icon_rect(r)
    icon = index.data(Qt.ItemDataRole.DecorationRole)
    if icon and not icon.isNull():
        icon.paint(painter, ir)
    else:
        painter.fillRect(ir, QColor("#1a2535"))
        painter.setPen(QColor("#2a3a4a"))
        painter.drawRect(ir)

    # — star overlay (top-left of poster) —————
    is_fav = bool(index.data(FAV_ROLE))
    sr = self.star_rect(r)

    # semi-transparent black background pill
    painter.setBrush(QColor(0, 0, 0, 140))
    painter.setPen(Qt.PenStyle.NoPen)
    painter.drawRoundedRect(sr, 4, 4)
```

```
# star character
fs = QFont()
fs.setPixelSize(_STAR_SIZE - 4)
painter.setFont(fs)
painter.setPen(QColor("#f9a825") if is_fav else QColor(255, 255, 255, 160))
painter.drawText(sr, Qt.AlignmentFlag.AlignCenter, "*" if is_fav else "x")

painter.restore()

def sizeHint(self, option, index):
    return QSize(self.poster_w + 14, self.poster_h + 6)
_PAD = 8
_ROW_H = 66

_CH_STAR = 14 # star overlay size for channel logos

class ChannelDelegate(QStyledItemDelegate):

    def __init__(self, parent=None):
        super().__init__(parent)
        self.select_mode: bool = False

    @staticmethod
    def star_rect_for(icon_rect: QRect) -> QRect:
        """Return the star hit/draw area for a given icon rect."""
        return QRect(icon_rect.left() + 1, icon_rect.top() + 1, _CH_STAR, _CH_STAR)

    def paint(self, painter: QPainter, option, index):
        painter.save()

        selected = bool(option.state & QStyle.StateFlag.State_Selected)
        hovered = bool(option.state & QStyle.StateFlag.State_MouseOver)

        bg = QColor("#1565c0") if selected else (QColor("#1c1c1c") if hovered else QColor("#111111"))
        painter.fillRect(option.rect, bg)

        painter.setPen(QPen(QColor("#1e1e1e")))
        painter.drawLine(option.rect.bottomLeft(), option.rect.bottomRight())

        r = option.rect

        # — icon —————
        ix = r.left() + _PAD
        iy = r.top() + (r.height() - _ICON_SIZE) // 2 - 4
        icon_rect = QRect(ix, iy, _ICON_SIZE, _ICON_SIZE)
        is_radio = bool(index.data(RADIO_ROLE))
        icon = index.data(Qt.ItemDataRole.DecorationRole)
        if icon and not icon.isNull():
            icon.paint(painter, icon_rect)
        else:
            ch_name = index.data(Qt.ItemDataRole.DisplayRole) or ""
            fallback = make_radio_avatar(_ICON_SIZE) if is_radio else make_avatar(ch_name, _ICON_SIZE)
            painter.drawPixmap(icon_rect, fallback)

        # — unreliable-stream amber ring —————
        if bool(index.data(FAIL_ROLE)):
            painter.setRenderHint(QPainter.RenderHint.Antialiasing)
            painter.setBrush(Qt.BrushStyle.NoBrush)
```

```

        painter.setPen(QPen(QColor("#f9a825"), 2))
        painter.drawRoundedRect(icon_rect.adjusted(-2, -2, 2, 2), 5, 5)

# — star overlay on logo (top-left) —————
is_fav = bool(index.data(FAV_ROLE))
sr = ChannelDelegate.star_rect_for(icon_rect)
painter.setRenderHint(QPainter.RenderHint.Antialiasing)
painter.setBrush(QColor(0, 0, 0, 140))
painter.setPen(Qt.PenStyle.NoPen)
painter.drawRoundedRect(sr, 3, 3)
fs = QFont(); fs.setPixelSize(10)
painter.setFont(fs)
painter.setPen(QColor("#f9a825") if is_fav else QColor(255, 255, 255, 160))
painter.drawText(sr, Qt.AlignmentFlag.AlignCenter, "*" if is_fav else "☆")

tx = icon_rect.right() + _PAD

# — checkbox (shown in multi-select mode) —————
_CB = 18
is_checkable = self.select_mode
if is_checkable:
    checked = int(index.data(Qt.ItemDataRole.CheckStateRole) or 0) == 2
    cb_x = r.right() - _CB - _PAD
    cb_y = r.top() + (r.height() - _CB) // 2
    painter.setRenderHint(QPainter.RenderHint.Antialiasing)
    painter.setBrush(QColor("#1976d2") if checked else QColor("#0a0f1a"))
    painter.setPen(QPen(QColor("#1976d2" if checked else "#4a6a8a"), 1.5))
    painter.drawRoundedRect(cb_x, cb_y, _CB, _CB, 3, 3)
    if checked:
        painter.setPen(QPen(QColor("white"), 2.0))
        painter.drawLine(cb_x + 3, cb_y + 10, cb_x + 7, cb_y + 13)
        painter.drawLine(cb_x + 7, cb_y + 13, cb_x + 15, cb_y + 5)

tw = r.right() - tx - _PAD - (_CB + _PAD if is_checkable else 0)

# — quality badge (colour-coded by quality level) —————
quality = "" if is_radio else (index.data(QUALITY_ROLE) or "")
badge_w = 0
if quality:
    fq = QFont(); fq.setPixelSize(9); fq.setBold(True)
    painter.setFont(fq)
    fm_q = painter.fontMetrics()
    badge_w = fm_q.horizontalAdvance(quality) + 10
    badge_h = 15
    badge_x = r.right() - badge_w - _PAD
    badge_y = r.top() + 5
    _QCOLORS = {
        "4K": "#6a1b9a",
        "1080p": "#1565c0",
        "720p": "#00695c",
        "480p": "#4a4a4a",
        "SD": "#4a4a4a",
    }
    bg = _QCOLORS.get(quality, "#2a3a4a")
    painter.setBrush(QColor(bg))
    painter.setPen(Qt.PenStyle.NoPen)
    painter.setRenderHint(QPainter.RenderHint.Antialiasing)
    painter.drawRoundedRect(badge_x, badge_y, badge_w, badge_h, 4, 4)
    painter.setPen(QColor("ffffff"))
    painter.drawText(QRect(badge_x, badge_y, badge_w, badge_h),

```

```

        Qt.AlignmentFlag.AlignCenter, quality)

# — channel name —————
is_fav = bool(index.data(FAV_ROLE))
fn = QFont(); fn.setPixelSize(13)
painter.setFont(fn)
name_color = "#f9a825" if is_fav else ("#ffffff" if selected else "#dddddd")
painter.setPen(QColor(name_color))
ch_name = index.data(Qt.ItemDataRole.DisplayRole) or ""
display_name = ch_name
name_w = tw - badge_w - (6 if badge_w else 0)
painter.drawText(QRect(tx, r.top() + 6, name_w, 17),
                 Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignVCenter,
                 display_name)

# — EPG subtitle —————
fe = QFont(); fe.setPixelSize(11)
painter.setFont(fe)
painter.setPen(QColor("#c0cce0" if selected else "#555555"))
painter.drawText(QRect(tx, r.top() + 25, tw, 15),
                 Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignVCenter,
                 index.data(EPG_ROLE) or "")

# — progress bar —————
prog_data = index.data(EPG_PROG_ROLE)
if prog_data:
    try:
        start_dt = datetime.fromtimestamp(prog_data[0])
        stop_dt = datetime.fromtimestamp(prog_data[1])
        now_dt = datetime.now()
        total = (stop_dt - start_dt).total_seconds()
        elapsed = (now_dt - start_dt).total_seconds()

        if total > 0 and elapsed > 0:
            # If elapsed overruns (EPG stale / channel loops), wrap into
            # the current airing so the bar and time remain meaningful.
            if elapsed > total:
                elapsed = elapsed % total
            pct = max(0.0, min(1.0, elapsed / total))
            bar_x = tx
            bar_y = r.top() + 45
            bar_w = tw
            bar_h = 5

            # background (remaining)
            painter.fillRect(QRect(bar_x, bar_y, bar_w, bar_h),
                            QColor("#333333"))

            # filled (elapsed)
            fill_w = max(1, int(bar_w * pct))
            painter.fillRect(QRect(bar_x, bar_y, fill_w, bar_h),
                            QColor("#42a5f5"))

            # elapsed label
            m, _ = divmod(int(elapsed), 60)
            h, m = divmod(m, 60)
            label = f"{h}h {m}m in" if h else f"{m}m in"
            ft = QFont(); ft.setPixelSize(10)
            painter.setFont(ft)
            painter.setPen(QColor("#c0cce0" if selected else "#888888"))

```

```

        painter.drawText(QRect(bar_x, bar_y + 6, bar_w, 12),
                        Qt.AlignmentFlag.AlignRight | Qt.AlignmentFlag.AlignVCenter,
                        label)

    except Exception:
        pass

# — radio badge —————
if index.data(RADIO_ROLE):
    badge_w, badge_h = 20, 12
    bx = icon_rect.right() - badge_w + 2
    by = icon_rect.bottom() - badge_h + 2
    painter.setBrush(QColor("#7b1fa2"))
    painter.setPen(Qt.PenStyle.NoPen)
    painter.drawRoundedRect(bx, by, badge_w, badge_h, 3, 3)
    fb = QFont(); fb.setPixelSize(8); fb.setBold(True)
    painter.setFont(fb)
    painter.setPen(QColor("white"))
    painter.drawText(QRect(bx, by, badge_w, badge_h),
                    Qt.AlignmentFlag.AlignCenter, "RD0")

painter.restore()

def sizeHint(self, option, index):
    return QSize(option.rect.width(), _ROW_H)

```

lxplayer/login_dialog.py

```

from requests.utils import urlparse

from PyQt6.QtWidgets import (
    QDialog, QVBoxLayout, QFormLayout, QHBoxLayout, QLabel,
    QLineEdit, QPushButton, QWidget, QFrame,
)
from PyQt6.QtCore import Qt
from PyQt6.QtGui import QFont

from lxplayer.constants import SERVER as _SERVER

class LoginDialog(QDialog):
    def __init__(self, config, parent=None):
        super().__init__(parent)
        self.config = config
        self.setWindowTitle("TVS Hub - Connect")
        self.setFixedWidth(460)
        self.setModal(True)
        self._build_ui()
        self._prefill()

    def _build_ui(self):
        layout = QVBoxLayout(self)
        layout.setSpacing(12)
        layout.setContentsMargins(32, 28, 32, 28)

        title = QLabel("TVS Hub")
        title.setAlignment(Qt.AlignmentFlag.AlignCenter)
        title.setStyleSheet("color: #1976d2; font-size: 22px; font-weight: bold;")
        layout.addWidget(title)

```

```
sub = QLabel("Enter your credentials")
sub.setAlignment(Qt.AlignmentFlag.AlignCenter)
sub.setStyleSheet("color: #666; font-size: 12px;")
layout.addWidget(sub)

# — saved accounts —————
if self.config.accounts:
    sep = QFrame()
    sep.setFrameShape(QFrame.Shape.HLine)
    sep.setStyleSheet("color: #2a2a2a;")
    layout.addWidget(sep)

    saved_lbl = QLabel("Saved Accounts")
    saved_lbl.setStyleSheet("color: #888; font-size: 11px; font-weight: bold;")
    layout.addWidget(saved_lbl)

    for acc in self.config.accounts:
        label = acc.label or acc.username
        btn = QPushButton(label)
        btn.setStyleSheet(
            "QPushButton{background:#1a2535;color:#d0d8e8;border:1px solid #1e2a3a;"
            "border-radius:5px;padding:6px 12px;text-align:left;font-size:12px;}"
            "QPushButton:hover{background:#1e3048;border-color:#1976d2;}"
        )
        btn.clicked.connect(lambda _, a=acc: self._load_account(a))
        layout.addWidget(btn)

    sep2 = QFrame()
    sep2.setFrameShape(QFrame.Shape.HLine)
    sep2.setStyleSheet("color: #2a2a2a;")
    layout.addWidget(sep2)

# — credential form —————
form = QFormLayout()
form.setSpacing(10)
form.setLabelAlignment(Qt.AlignmentFlag.AlignRight)

self.user_edit = QLineEdit()
self.user_edit.setPlaceholderText("username")
self.user_edit.setFixedHeight(34)
form.addRow("Username:", self.user_edit)

pass_row = QWidget()
pass_row.setStyleSheet("background: transparent;")
pl = QHBoxLayout(pass_row)
pl.setContentsMargins(0, 0, 0, 0)
pl.setSpacing(4)

self.pass_edit = QLineEdit()
self.pass_edit.setPlaceholderText("password")
self.pass_edit.setEchoMode(QLineEdit.EchoMode.Password)
self.pass_edit.setFixedHeight(34)
pl.addWidget(self.pass_edit)

self.eye_btn = QPushButton("")
self.eye_btn.setFixedSize(34, 34)
self.eye_btn.setCheckable(True)
self.eye_btn.setToolTip("Show / hide password")
self.eye_btn.setStyleSheet("""
```

```

        QPushButton {
            background: #1a1a1a; border: 1px solid #2a2a2a;
            border-radius: 5px; color: #666; font-size: 15px;
        }
        QPushButton:hover { color: #fff; border-color: #444; }
        QPushButton:checked { color: #1976d2; border-color: #1976d2; }
    """
    self.eye_btn.toggled.connect(
        lambda on: self.pass_edit.setEchoMode(
            QLineEdit.EchoMode.Normal if on else QLineEdit.EchoMode.Password
        )
    )
    pl.addWidget(self.eye_btn)
    form.addRow("Password:", pass_row)

    layout.addLayout(form)
    layout.addStretch()

    self.connect_btn = QPushButton("Connect")
    self.connect_btn.setFixedHeight(42)
    self.connect_btn.setStyleSheet("""
        QPushButton {
            background-color: #1976d2; color: white;
            border: none; border-radius: 6px;
            font-size: 14px; font-weight: bold;
        }
        QPushButton:hover { background-color: #1e88e5; }
        QPushButton:pressed { background-color: #1565c0; }
    """)
    self.connect_btn.clicked.connect(self._on_connect)
    self.pass_edit.returnPressed.connect(self._on_connect)
    layout.addWidget(self.connect_btn)

    def _prefill(self):
        acc = self.config.account
        self.user_edit.setText(acc.username)
        self.pass_edit.setText(acc.password)

    def _load_account(self, acc):
        self.user_edit.setText(acc.username)
        self.pass_edit.setText(acc.password)

    def _on_connect(self):
        if self.user_edit.text().strip():
            self.accept()

    def credentials(self) -> tuple[str, str, str]:
        return (
            _SERVER,
            self.user_edit.text().strip(),
            self.pass_edit.text().strip(),
        )

```

lxplayer/log_panel.py

```

from PyQt6.QtWidgets import (
    QDialog, QVBoxLayout, QHBoxLayout,
    QTextEdit, QPushButton, QLabel,

```



```
)  
from PyQt6.QtGui import QFont  
from PyQt6.QtCore import Qt  
  
_COLORS = {  
    "INFO": "#cccccc",  
    "REQ": "#64b5f6",  
    "RESP": "#a5d6a7",  
    "ERROR": "#ef9a9a",  
}  
  
class LogPanel(QDialog):  
    def __init__(self, parent=None):  
        super().__init__(parent)  
        self.setWindowTitle("TVS Hub - Debug Log")  
        self.resize(740, 420)  
        self.setStyleSheet("""  
            QDialog { background: #0d0d0d; }  
            QPushButton {  
                background: #1e1e1e;  
                color: #aaa;  
                border: 1px solid #333;  
                border-radius: 4px;  
                padding: 5px 12px;  
                font-size: 12px;  
            }  
            QPushButton:hover { background: #2a2a2a; color: white; }  
        """)  
        self._build_ui()  
  
    def _build_ui(self):  
        layout = QVBoxLayout(self)  
        layout.setContentsMargins(10, 10, 10, 10)  
        layout.setSpacing(8)  
  
        header = QHBoxLayout()  
        lbl = QLabel("Debug Log")  
        lbl.setStyleSheet("color: #666; font-size: 11px; font-weight: bold;")  
        header.addWidget(lbl)  
        header.addStretch()  
  
        legend = QHBoxLayout()  
        legend.setSpacing(14)  
        for level, color in _COLORS.items():  
            dot = QLabel(f"● {level}")  
            dot.setStyleSheet(f"color: {color}; font-size: 11px;")  
            legend.addWidget(dot)  
        header.addLayout(legend)  
  
        clear_btn = QPushButton("Clear")  
        clear_btn.clicked.connect(self._log.clear if hasattr(self, "_log") else lambda: None)  
        header.addWidget(clear_btn)  
        layout.addLayout(header)  
  
        self._log = QTextEdit()  
        self._log.setReadOnly(True)  
        self._log.setFont(QFont("Courier New, Consolas, Monospace", 10))  
        self._log.setStyleSheet("""  
            QTextEdit {
```

```

        background: #050505;
        color: #ccc;
        border: 1px solid #1e1e1e;
        border-radius: 4px;
        padding: 6px;
    }
    """
    layout.addWidget(self._log)

    clear_btn.clicked.connect(self._log.clear)

    btn_row = QHBoxLayout()
    btn_row.addStretch()
    close_btn = QPushButton("Close")
    close_btn.clicked.connect(self.hide)
    btn_row.addWidget(close_btn)
    layout.addLayout(btn_row)

def append(self, level: str, message: str):
    color = _COLORS.get(level, "#ccc")
    # Preserve indentation by replacing spaces with &nbsp; only for continuation lines
    lines = message.split("\n")
    html_lines = []
    for i, line in enumerate(lines):
        escaped = (line.replace("&", "&amp;")
                  .replace("<", "&lt;")
                  .replace(">", "&gt;")
                  .replace(" ", "&nbsp;"))

        if i == 0:
            html_lines.append(f'<span style="color:{color}">{escaped}</span>')
        else:
            html_lines.append(f'<span style="color:#666">{escaped}</span>')
    self._log.append("<br>".join(html_lines))
    sb = self._log.verticalScrollBar()
    sb.setValue(sb.maximum())

```

lxplayer/home_screen.py

```

from datetime import datetime

from PyQt6.QtWidgets import QWidget, QVBoxLayout, QHBoxLayout, QLabel, QPushButton
from PyQt6.QtCore import Qt, QTimer, pyqtSignal, QRectF
from PyQt6.QtGui import QPainter, QPainterPath, QLinearGradient, QColor, QFont, QFontMetrics

class _BrandWidget(QWidget):
    """TVS Hub' gradient logo."""

    def __init__(self, parent=None):
        super().__init__(parent)
        self._lf = QFont()
        self._lf.setPixelSize(28)
        self._lf.setBold(True)
        self._lf.setLetterSpacing(QFont.SpacingType.AbsoluteSpacing, 1.5)

        self._lw = QFontMetrics(self._lf).horizontalAdvance("TVS Hub")

        self.setFixedSize(self._lw + 4, 44)

```

```
self.setAttribute(Qt.WidgetAttribute.WA_TranslucentBackground)

def paintEvent(self, _):
    p = QPainter(self)
    p.setRenderHint(QPainter.RenderHint.Antialiasing)
    p.setRenderHint(QPainter.RenderHint.TextAntialiasing)

    # gradient "TVS Hub"
    path = QPainterPath()
    path.addText(2, 30, self._lf, "TVS Hub")
    grad = QLinearGradient(0, 0, self._lw, 0)
    grad.setColorAt(0.0, QColor("#1565c0"))
    grad.setColorAt(0.45, QColor("#42a5f5"))
    grad.setColorAt(1.0, QColor("#00e5ff"))
    p.fillPath(path, grad)
    p.end()

class GradientCard(QWidget):
    """Large section card with diagonal gradient background."""
    clicked = pyqtSignal()

    def __init__(self, c1: str, c2: str, icon: str, label: str, parent=None):
        super().__init__(parent)
        self._c1 = QColor(c1)
        self._c2 = QColor(c2)
        self._hover = False
        self.setCursor(Qt.CursorShape.PointingHandCursor)

        vl = QVBoxLayout(self)
        vl.setContentsMargins(20, 28, 20, 20)
        vl.setSpacing(10)
        vl.addStretch()

        icon_lbl = QLabel(icon)
        icon_lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
        icon_lbl.setStyleSheet("background:transparent; color:white; font-size:46px;")
        vl.addWidget(icon_lbl)

        title_lbl = QLabel(label)
        title_lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
        f = QFont()
        f.setPixelSize(17)
        f.setBold(True)
        title_lbl.setFont(f)
        title_lbl.setStyleSheet("background:transparent; color:white; letter-spacing:2px;")
        vl.addWidget(title_lbl)

        vl.addStretch()

    def paintEvent(self, event):
        p = QPainter(self)
        p.setRenderHint(QPainter.RenderHint.Antialiasing)
        path = QPainterPath()
        path.addRoundedRect(0.0, 0.0, float(self.width()), float(self.height()), 14.0, 14.0)
        c1 = self._c1.lighter(112) if self._hover else self._c1
        c2 = self._c2.lighter(112) if self._hover else self._c2
        grad = QLinearGradient(0, 0, self.width(), self.height())
        grad.setColorAt(0.0, c1)
        grad.setColorAt(1.0, c2)
```

```
p.fillPath(path, grad)

def enterEvent(self, event):
    self._hover = True
    self.update()

def leaveEvent(self, event):
    self._hover = False
    self.update()

def mousePressEvent(self, event):
    if event.button() == Qt.MouseButton.LeftButton:
        self.clicked.emit()

class SmallCard(QWidget):
    """Compact dark card for secondary features."""
    clicked = pyqtSignal()

    def __init__(self, icon: str, label: str, active: bool = False, parent=None):
        super().__init__(parent)
        self._active = active
        self._hover = False
        self.setCursor(Qt.CursorShape.PointingHandCursor)

        hl = QHBoxLayout(self)
        hl.setContentsMargins(16, 14, 16, 14)
        hl.setSpacing(12)

        icon_lbl = QLabel(icon)
        icon_lbl.setFixedWidth(26)
        icon_lbl.setStyleSheet("background:transparent; color:white; font-size:20px;")
        hl.addWidget(icon_lbl)

        text_lbl = QLabel(label)
        f = QFont()
        f.setPixelSize(12)
        f.setBold(True)
        text_lbl.setFont(f)
        text_lbl.setStyleSheet("background:transparent; color:white; letter-spacing:1px;")
        hl.addWidget(text_lbl)
        hl.addStretch()

    def paintEvent(self, event):
        p = QPainter(self)
        p.setRenderHint(QPainter.RenderHint.Antialiasing)
        path = QPainterPath()
        path.addRoundedRect(0.0, 0.0, float(self.width()), float(self.height()), 10.0, 10.0)
        base = QColor("#1b3a4b") if self._active else QColor("#162230")
        color = base.lighter(115) if self._hover else base
        p.fillPath(path, color)

    def enterEvent(self, event):
        self._hover = True
        self.update()

    def leaveEvent(self, event):
        self._hover = False
        self.update()
```

```
def mousePressEvent(self, event):
    if event.button() == Qt.MouseButton.LeftButton:
        self.clicked.emit()

class HomeScreen(QWidget):
    # 0 = Live TV, 1 = Movies, 2 = Series, 3 = Radio
    section_clicked = pyqtSignal(int)
    login_clicked = pyqtSignal()
    logoff_clicked = pyqtSignal()
    epq_clicked = pyqtSignal()
    multiscreen_clicked = pyqtSignal()
    catchup_clicked = pyqtSignal()
    radio_clicked = pyqtSignal()
    settings_clicked = pyqtSignal()
    backup_clicked = pyqtSignal()

    def __init__(self, parent=None):
        super().__init__(parent)
        self.setStyleSheet("background: #0d1b2a;")
        self._build_ui()

        self._timer = QTimer(self)
        self._timer.timeout.connect(self._tick)
        self._timer.start(1000)
        self._tick()

# — public —————

def set_expiry(self, text: str):
    self._expiry_lbl.setText(f"Expiration : {text}")

def warn_expiry(self, days: int):
    if days <= 0:
        color, suffix = "#e53935", " (EXPIRED)"
    elif days <= 3:
        color, suffix = "#e53935", f" ({days}d left!)"
    else:
        color, suffix = "#fb8c00", f" ({days}d left)"
    txt = self._expiry_lbl.text()
    if "left" not in txt and "EXPIRED" not in txt:
        self._expiry_lbl.setText(txt + suffix)
    self._expiry_lbl.setStyleSheet(
        f"color:{color}; font-size:13px; font-weight:bold;"
    )

def set_username(self, name: str):
    self._user_btn.setText(name)
    self._logoff_btn.show()

def set_logged_out(self):
    self._user_btn.setText("Login")
    self._logoff_btn.hide()
    self._expiry_lbl.setText("Expiration : -")

# — internals —————

def _tick(self):
    now = datetime.now()
    self._time_lbl.setText(now.strftime("%I:%M %p"))
```

```
self._date_lbl.setText(now.strftime("%b %d, %Y"))

def _build_ui(self):
    vl = QVBoxLayout(self)
    vl.setContentsMargins(28, 18, 28, 18)
    vl.setSpacing(0)

    vl.addWidget(self._build_topbar())
    vl.addSpacing(28)
    vl.addLayout(self._build_cards(), stretch=1)
    vl.addSpacing(18)

    self._expiry_lbl = QLabel("Expiration : -")
    self._expiry_lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
    self._expiry_lbl.setStyleSheet("color:#4a6fa5; font-size:13px;")
    vl.addWidget(self._expiry_lbl)

    credit = QLabel("Created by David Mills")
    credit.setAlignment(Qt.AlignmentFlag.AlignCenter)
    credit.setStyleSheet("color:#4a6fa5; font-size:10px;")
    credit.hide()
    vl.addWidget(credit)

    bottom = QHBoxLayout()
    bottom.setContentsMargins(0, 0, 0, 0)
    ver = QLabel("v1.0.1")
    ver.setStyleSheet("color:#4a6a8a; font-size:9px;")
    bottom.addWidget(ver)
    bottom.addStretch()
    dm2 = QLabel("DM2")
    dm2.setStyleSheet("color:#4a6a8a; font-size:9px;")
    bottom.addWidget(dm2)
    vl.addLayout(bottom)

def _build_topbar(self) -> QWidget:
    bar = QWidget()
    bar.setFixedHeight(56)
    bar.setStyleSheet("background: transparent;")
    hl = QHBoxLayout(bar)
    hl.setContentsMargins(0, 0, 0, 0)
    hl.setSpacing(0)

    hl.addWidget(_BrandWidget())

    hl.addStretch()

    self._time_lbl = QLabel("")
    f2 = QFont()
    f2.setPixelSize(20)
    f2.setBold(True)
    self._time_lbl.setFont(f2)
    self._time_lbl.setStyleSheet("color: white;")
    hl.addWidget(self._time_lbl)

    self._date_lbl = QLabel("")
    self._date_lbl.setStyleSheet("color: #7a9cc0; font-size: 13px; margin-left: 14px;")
    hl.addWidget(self._date_lbl)

    hl.addStretch()
```

```
_pill = (
    "QPushButton { background: #1976d2; color: white; border: none;"
    " border-radius: 4px; font-size: 11px; font-weight: bold; padding: 5px 10px; }"
    "QPushButton:hover { background: #1e88e5; }"
)

self._user_btn = QPushButton("Login")
self._user_btn.setStyleSheet(_pill)
self._user_btn.clicked.connect(self.login_clicked)
hl.addWidget(self._user_btn)

self._logoff_btn = QPushButton("Log Off")
self._logoff_btn.setStyleSheet(
    "QPushButton { background: #c62828; color: white; border: none;"
    " border-radius: 4px; font-size: 11px; font-weight: bold; padding: 5px 10px; }"
    "QPushButton:hover { background: #e53935; }"
)
self._logoff_btn.clicked.connect(self.logoff_clicked)
self._logoff_btn.hide()
hl.addWidget(self._logoff_btn)

gear_btn = QPushButton("⚙")
gear_btn.setFixedSize(32, 32)
gear_btn.setToolTip("Player Settings")
gear_btn.setStyleSheet(
    "QPushButton{background:transparent;color:#7a9cc0;border:none;"
    "font-size:18px;margin-left:6px;}"
    "QPushButton:hover{color:white;}"
)
gear_btn.clicked.connect(self.settings_clicked)
hl.addWidget(gear_btn)

backup_btn = QPushButton("")
backup_btn.setFixedSize(32, 32)
backup_btn.setToolTip("Backup & Restore")
backup_btn.setStyleSheet(
    "QPushButton{background:transparent;color:#7a9cc0;border:none;"
    "font-size:18px;margin-left:2px;}"
    "QPushButton:hover{color:white;}"
)
backup_btn.clicked.connect(self.backup_clicked)
hl.addWidget(backup_btn)

return bar

def _build_cards(self) -> QHBoxLayout:
    root = QHBoxLayout()
    root.setSpacing(16)

    # — LIVE TV & RADIO (tall card, left) —————
    live = GradientCard("#1fc8a0", "#0968c3", "", "LIVE TV")
    live.setMinimumWidth(200)
    live.clicked.connect(lambda: self.section_clicked.emit(0))
    root.addWidget(live, stretch=5)

    # — Right column —————
    right = QVBoxLayout()
    right.setSpacing(16)

    # Top row: Movies + Series
    top = QHBoxLayout()
```

```
top.setSpacing(16)

movies = GradientCard("#ff512f", "#c0116b", "▶", "MOVIES")
movies.clicked.connect(lambda: self.section_clicked.emit(1))
top.addWidget(movies, stretch=1)

series = GradientCard("#8e2de2", "#4568dc", "", "SERIES")
series.clicked.connect(lambda: self.section_clicked.emit(2))
top.addWidget(series, stretch=1)
right.addLayout(top, stretch=6)

# Bottom row: secondary features
bot = QHBoxLayout()
bot.setSpacing(16)
for icon, label in [
    ("", "LIVE WITH EPG"),
    ("⌘", "MULTI-SCREEN"),
    ("", "CATCH UP"),
    ("", "RADIO"),
]:
    card = SmallCard(icon, label)
    if label == "LIVE WITH EPG":
        card.clicked.connect(self.epg_clicked)
    elif label == "MULTI-SCREEN":
        card.clicked.connect(self.multiscreen_clicked)
    elif label == "CATCH UP":
        card.clicked.connect(self.catchup_clicked)
    elif label == "RADIO":
        card.clicked.connect(lambda: self.section_clicked.emit(3))
    bot.addWidget(card, stretch=1)
right.addLayout(bot, stretch=3)

root.addLayout(right, stretch=7)
return root
```

lxplayer/epg_screen.py

```
import base64
from datetime import datetime, timedelta

from PyQt6.QtWidgets import (
    QWidget, QVBoxLayout, QHBoxLayout, QLabel, QPushButton,
    QScrollArea, QGridLayout, QStackedWidget, QDialog,
)
from PyQt6.QtCore import Qt, QTimer, pyqtSignal, QThreadPool
from PyQt6.QtGui import QPainter, QColor, QFont, QPainterPath, QPixmap

from .player import PlayerWidget
from .workers import Worker
from .utils import make_avatar, fetch_logo_bytes, round_pixmap

_BG = "#0a0f1a"
_PANEL = "#0d1117"
_BORDER = "#1e2a3a"
_TEXT = "#d0d8e8"
_DIM = "#4a6a8a"
_SEL = "#1565c0"
_TEAL = "#1fc8a0"
```



```
ROW_H          = 52
HDR_H          = 34
MINS_W         = 4      # pixels per minute
CH_W           = 230    # channel name column width
MAX_EPG_CHANNELS = 500  # X11 widget height limit is 32767px; 500 rows = 26034px

def _decode(raw: str) -> str:
    if not raw:
        return ""
    try:
        return base64.b64decode(raw + "==").decode("utf-8")
    except Exception:
        return raw

def _hdr_btn(text: str) -> QPushButton:
    b = QPushButton(text)
    b.setFixedSize(32, 32)
    b.setStyleSheet(
        "QPushButton{background:transparent;color:#42a5f5;border:none;font-size:18px;}"
        "QPushButton:hover{color:white;}"
    )
    return b

# — category row widget —————

class _CatRow(QWidget):
    clicked = pyqtSignal()

    def __init__(self, name: str, color: str = None, parent=None):
        super().__init__(parent)
        self._hover = False
        self.setCursor(Qt.CursorShape.PointingHandCursor)
        self.setFixedHeight(48)
        hl = QHBoxLayout(self)
        hl.setContentsMargins(14, 0, 14, 0)
        hl.setSpacing(10)

        ico = QLabel("▶")
        ico.setFixedWidth(18)
        ico.setStyleSheet(f"color:{color or _TEAL};font-size:12px;background:transparent;")
        hl.addWidget(ico)

        lbl = QLabel(name)
        lbl.setStyleSheet(f"color:{color or _TEXT};font-size:13px;font-weight:bold;background:transparent;")
        hl.addWidget(lbl, stretch=1)

        arr = QLabel(">")
        arr.setStyleSheet(f"color:{_DIM};font-size:20px;background:transparent;")
        hl.addWidget(arr)

    def paintEvent(self, _):
        p = QPainter(self)
        p.setRenderHint(QPainter.RenderHint.Antialiasing)
        path = QPainterPath()
        path.addRoundedRect(4.0, 2.0, float(self.width() - 8), float(self.height() - 4), 8.0, 8.0)
        p.fillPath(path, QColor("#1a2d40" if self._hover else "#162230"))
```

```
def enterEvent(self, _): self._hover = True; self.update()
def leaveEvent(self, _): self._hover = False; self.update()
def mousePressEvent(self, e):
    if e.button() == Qt.MouseButton.LeftButton:
        self.clicked.emit()

# — EPG category list screen (epgl.png) —————

class EPGCategoryScreen(QWidget):
    category_selected = pyqtSignal(object, str) # (cat_id or None, display_name)
    back_clicked      = pyqtSignal()

    def __init__(self, title: str = "EPG Categories", parent=None):
        super().__init__(parent)
        self._title = title
        self.setStyleSheet(f"background:{_BG};")
        self._build()

    def _build(self):
        vl = QVBoxLayout(self)
        vl.setContentsMargins(0, 0, 0, 0)
        vl.setSpacing(0)

        hdr = QWidget()
        hdr.setFixedHeight(52)
        hdr.setStyleSheet(f"background:{_PANEL};border-bottom:1px solid {_BORDER};")
        hl = QHBoxLayout(hdr)
        hl.setContentsMargins(16, 0, 16, 0)
        hl.setSpacing(12)
        back = _hdr_btn("<-")
        back.clicked.connect(self.back_clicked)
        hl.addWidget(back)
        title = QLabel(self._title)
        title.setStyleSheet("color:white;font-size:15px;font-weight:bold;background:transparent;")
        hl.addWidget(title, stretch=1)
        vl.addWidget(hdr)

        scroll = QScrollArea()
        scroll.setWidgetResizable(True)
        scroll.setStyleSheet("QScrollArea{border:none;background:transparent;}")
        self._inner = QWidget()
        self._inner.setStyleSheet(f"background:{_BG};")
        self._grid = QGridLayout(self._inner)
        self._grid.setContentsMargins(16, 16, 16, 16)
        self._grid.setSpacing(8)
        self._grid.setColumnStretch(0, 1)
        self._grid.setColumnStretch(1, 1)
        scroll.setWidget(self._inner)
        vl.addWidget(scroll, stretch=1)

    def load_categories(self, cats: list):
        while self._grid.count():
            w = self._grid.takeAt(0)
            if w.widget():
                w.widget().deleteLater()

        _SPECIAL = ("__FAVORITES__", "__RECENT__")
        items = []
```

```

# pinned specials first
for c in cats:
    if c.get("category_id") in _SPECIAL:
        items.append((c.get("category_name", "-"), c.get("category_id"), c.get("_color")))
# ALL
items.append(("ALL", None, None))
# regular categories
for c in cats:
    if c.get("category_id") not in _SPECIAL:
        items.append((c.get("category_name", "-"), c.get("category_id"), None))

for i, (name, cid, color) in enumerate(items):
    row, col = divmod(i, 2)
    r = _CatRow(name, color=color)
    r.clicked.connect(lambda c=cid, n=name: self.category_selected.emit(c, n))
    self.grid.addWidget(r, row, col)

if len(items) % 2 == 1:
    self.grid.addWidget(QWidget(), len(items) // 2, 1)

# — timeline painter —————

class _TimelineWidget(QWidget):
    row_clicked = pyqtSignal(int)
    prog_clicked = pyqtSignal(dict)

    _AV_SZ = 28 # avatar size inside the timeline channel column

    def __init__(self):
        super().__init__()
        self._rows: list[dict] = []
        self._start: datetime | None = None
        self._sel = -1
        self._avatars: list = []

    def set_data(self, rows: list, start: datetime):
        self._rows = rows
        self._start = start
        self._avatars = [make_avatar(r.get("name", ""), self._AV_SZ) for r in rows]
        self.setFixedSize(CH_W + 24 * 60 * MINS_W, HDR_H + max(1, len(rows)) * ROW_H)
        self.update()

    def update_avatar(self, idx: int, px: QPixmap):
        if 0 <= idx < len(self._avatars):
            self._avatars[idx] = px
            self.update()

    def select_row(self, idx: int):
        self._sel = idx
        self.update()

    def paintEvent(self, _):
        if not self._start:
            return
        p = QPainter(self)
        p.setRenderHint(QPainter.RenderHint.Antialiasing)
        now = datetime.now()
        W = self.width()

```

```

# time header
p.fillRect(0, 0, W, HDR_H, QColor(_PANEL))
p.fillRect(0, 0, CH_W, HDR_H, QColor("#0b1520"))
fn_hdr = QFont(); fn_hdr.setPixelSize(10); fn_hdr.setBold(True)
p.setFont(fn_hdr)
p.setPen(QColor(_DIM))
p.drawText(12, 0, CH_W - 12, HDR_H, Qt.AlignmentFlag.AlignVCenter, "CHANNEL")

fn_time = QFont(); fn_time.setPixelSize(11)
p.setFont(fn_time)
for h in range(25):
    x = CH_W + h * 60 * MINS_W
    dt = self._start + timedelta(hours=h)
    label = dt.strftime("%I:%M %p").lstrip("0") if dt.minute else dt.strftime("%I %p").lstrip("0")
    p.setPen(QColor(_DIM))
    p.drawText(x + 4, 0, 120, HDR_H, Qt.AlignmentFlag.AlignVCenter, label)
    p.setPen(QColor(_BORDER))
    p.drawLine(x, HDR_H - 8, x, HDR_H)

# rows
fn_ch = QFont(); fn_ch.setPixelSize(12)
fn_prg = QFont(); fn_prg.setPixelSize(11)
for i, row in enumerate(self._rows):
    y = HDR_H + i * ROW_H
    sel = (i == self._sel)

    bg = QColor(_SEL) if sel else (QColor("#0d1420") if i % 2 == 0 else QColor(_PANEL))
    p.fillRect(0, y, W, ROW_H, bg)

    fn_ch.setBold(sel)
    p.setFont(fn_ch)
    p.setPen(QColor("white") if sel else QColor(_TEXT))
    av_sz = self._AV_SZ
    av_x = 8
    av_y = y + (ROW_H - av_sz) // 2
    if i < len(self._avatars):
        p.drawPixmap(av_x, av_y, av_sz, av_sz, self._avatars[i])
    txt_x = av_x + av_sz + 8
    p.drawText(txt_x, y, CH_W - txt_x - 6, ROW_H, Qt.AlignmentFlag.AlignVCenter, row.get("name", ""))

    p.setPen(QColor(_BORDER))
    p.drawLine(CH_W, y, CH_W, y + ROW_H)
    p.drawLine(0, y + ROW_H - 1, W, y + ROW_H - 1)

    p.setFont(fn_prg)
    for prog in row.get("programs", []):
        try:
            ps = datetime.fromtimestamp(int(prog.get("start_timestamp", 0)))
            pe = datetime.fromtimestamp(int(prog.get("stop_timestamp", 0)))
        except Exception:
            continue
        dur = (pe - ps).total_seconds() / 60
        if dur <= 0:
            continue
        off = (ps - self._start).total_seconds() / 60
        bx = CH_W + int(off * MINS_W)
        bw = max(2, int(dur * MINS_W) - 2)
        by = y + 4
        bh = ROW_H - 8

```

```

        is_live = ps <= now <= pe
        blk_col = QColor("#0e4a7a" if is_live else ("#1a4878" if sel else "#1a3050"))
        path = QPainterPath()
        path.addRoundedRect(float(bx), float(by), float(bw), float(bh), 3.0, 3.0)
        p.fillPath(path, blk_col)
        p.setPen(QColor("#2a5a8a"))
        p.drawPath(path)

        title = _decode(prog.get("title", "")) or "No Information"
        p.setPen(QColor(_TEXT))
        fm = p.fontMetrics()
        p.drawText(bx + 5, by, bw - 10, bh - 5, Qt.AlignmentFlag.AlignVCenter,
                   fm.elidedText(title, Qt.TextElideMode.ElideRight, max(0, bw - 10)))

    # progress bar at bottom of live program block
    if is_live:
        prog_total = (pe - ps).total_seconds()
        if prog_total > 0:
            pct = max(0.0, min(1.0, (now - ps).total_seconds() / prog_total))
            bar_y = by + bh - 4
            # dark track = full duration
            p.fillRect(bx, bar_y, bw, 4, QColor("#333333"))
            # blue fill = elapsed
            fill_w = int(bw * pct)
            if fill_w > 0:
                p.fillRect(bx, bar_y, fill_w, 4, QColor("#42a5f5"))

    # current-time indicator
    off_now = (now - self._start).total_seconds() / 60
    if 0 <= off_now <= 24 * 60:
        nx = CH_W + int(off_now * MINS_W)
        p.setPen(QColor(_TEAL))
        p.drawLine(nx, 0, nx, self.height())

def mousePressEvent(self, e):
    if e.button() != Qt.MouseButton.LeftButton or not self._start:
        return
    row = (e.pos().y() - HDR_H) // ROW_H
    if not (0 <= row < len(self._rows)):
        return
    x = e.pos().x()
    if x < CH_W:
        self.row_clicked.emit(row)
        return
    # find which programme cell was clicked
    for prog in self._rows[row].get("programs", []):
        try:
            ps = datetime.fromtimestamp(int(prog.get("start_timestamp", 0)))
            pe = datetime.fromtimestamp(int(prog.get("stop_timestamp", 0)))
        except Exception:
            continue
        off = (ps - self._start).total_seconds() / 60
        dur = (pe - ps).total_seconds() / 60
        bx = CH_W + int(off * MINS_W)
        bw = max(2, int(dur * MINS_W))
        if bx <= x <= bx + bw:
            self.prog_clicked.emit({
                "title": _decode(prog.get("title", "")),
                "description": _decode(prog.get("description", "")),
                "start_timestamp": prog.get("start_timestamp", 0),

```

```
        "stop_timestamp": prog.get("stop_timestamp", 0),
    })
    return
self.row_clicked.emit(row)

# — Fullscreen player dialog —————

class _FullscreenPlayer(QDialog):
    def __init__(self, url: str, parent=None):
        super().__init__(parent)
        self._url = url
        self.setWindowFlags(
            Qt.WindowType.Window | Qt.WindowType.FramelessWindowHint
        )
        self.showFullScreen()

        vl = QVBoxLayout(self)
        vl.setContentsMargins(0, 0, 0, 0)
        vl.setSpacing(0)

        # thin back bar at top so the user always has a non-MPV click target
        bar = QWidget()
        bar.setFixedHeight(40)
        bar.setStyleSheet("background: rgba(0,0,0,200);")
        hl = QHBoxLayout(bar)
        hl.setContentsMargins(12, 0, 12, 0)
        back_btn = QPushButton("← Back to EPG")
        back_btn.setStyleSheet(
            "QPushButton{background:transparent;color:#42a5f5;border:none;font-size:13px;}"
            "QPushButton:hover{color:white;}"
        )
        back_btn.clicked.connect(self.accept)
        hl.addWidget(back_btn)
        hl.addStretch()
        hint = QLabel("click video · ESC")
        hint.setStyleSheet("color:#4a6a8a;font-size:11px;background:transparent;")
        hl.addWidget(hint)
        vl.addWidget(bar)

        self.player = PlayerWidget()
        self.player.clicked.connect(self.accept) # click video → close
        vl.addWidget(self.player, stretch=1)

        QTimer.singleShot(80, lambda: self.player.play(url))

    def keyPressEvent(self, e):
        if e.key() == Qt.Key.Key_Escape:
            self.accept()
        else:
            super().keyPressEvent(e)

    def closeEvent(self, e):
        self.player.stop()
        super().closeEvent(e)

# — EPG grid screen (epg2.png) —————

class EPGGridScreen(QWidget):
```

```
back_clicked      = pyqtSignal()
channel_activated = pyqtSignal(dict, str)

def __init__(self, api):
    super().__init__()
    self._api = api
    self._rows: list[dict] = []
    self._start: datetime | None = None
    self._epg_queue: list = []
    self._epg_active = 0
    self._epg_gen = 0
    self._icon_queue: list = []
    self._icon_active = 0
    self._icon_gen = 0
    self._current_url: str = ""
    self._day_offset = 0
    self._current_streams: list = []
    self._current_cat: str = ""
    self._scroll_to_now: bool = True
    self.setStyleSheet(f"background:{_BG};")
    self._build()

def _build(self):
    vl = QVBoxLayout(self)
    vl.setContentsMargins(0, 0, 0, 0)
    vl.setSpacing(0)

    # header bar
    hdr = QWidget()
    hdr.setFixedHeight(52)
    hdr.setStyleSheet(f"background:{_PANEL};border-bottom:1px solid {_BORDER};")
    hl = QHBoxLayout(hdr)
    hl.setContentsMargins(16, 0, 16, 0)
    hl.setSpacing(12)
    back = _hdr_btn("<")
    back.clicked.connect(self._on_back)
    hl.addWidget(back)
    self._cat_lbl = QLabel("")
    self._cat_lbl.setStyleSheet("color:white;font-size:15px;font-weight:bold;background:transparent;")
    hl.addWidget(self._cat_lbl, stretch=1)

    prev_day = _hdr_btn("<")
    prev_day.setToolTip("Previous day")
    prev_day.clicked.connect(lambda: self._shift_day(-1))
    hl.addWidget(prev_day)
    self._day_lbl = QLabel("Today")
    self._day_lbl.setStyleSheet(f"color:{_DIM};font-size:12px;background:transparent;min-width:80px;")
    self._day_lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
    hl.addWidget(self._day_lbl)
    next_day = _hdr_btn(">")
    next_day.setToolTip("Next day")
    next_day.clicked.connect(lambda: self._shift_day(1))
    hl.addWidget(next_day)

    self._clock = QLabel("")
    self._clock.setStyleSheet(f"color:{_DIM};font-size:12px;background:transparent;")
    hl.addWidget(self._clock)
    vl.addWidget(hdr)

    # top area: player + now-playing info
```

```
top = QWidget()
top.setFixedHeight(190)
top.setStyleSheet(f"background:{_PANEL};")
tl = QHBoxLayout(top)
tl.setContentsMargins(0, 0, 0, 0)
tl.setSpacing(0)

self.player = PlayerWidget()
self.player.setFixedWidth(300)
self.player.setCursor(Qt.CursorShape.PointingHandCursor)
self.player.clicked.connect(self._open_fullscreen)
tl.addWidget(self.player)

info_w = QWidget()
info_w.setStyleSheet(f"background:{_PANEL};border-left:1px solid {_BORDER};")
il = QVBoxLayout(info_w)
il.setContentsMargins(20, 14, 20, 14)
il.setSpacing(4)
self._prog_title = QLabel("Select a channel")
self._prog_title.setStyleSheet("color:white;font-size:14px;font-weight:bold;background:transparent;")
self._prog_title.setWordWrap(True)
il.addWidget(self._prog_title)
self._prog_time = QLabel("")
self._prog_time.setStyleSheet(f"color:{_DIM};font-size:12px;background:transparent;")
il.addWidget(self._prog_time)
il.addStretch()

fs_btn = QPushButton(" Fullscreen")
fs_btn.setStyleSheet(
    f"QPushButton{{background:transparent;color:{_DIM};border:none;"
    f"font-size:12px;text-align:left;}}"
    f"QPushButton:hover{{color:white;}}"
)
fs_btn.setCursor(Qt.CursorShape.PointingHandCursor)
fs_btn.clicked.connect(self._open_fullscreen)
il.addWidget(fs_btn)

tl.addWidget(info_w, stretch=1)
vl.addWidget(top)

# timeline
self._scroll = QScrollArea()
self._scroll.setWidgetResizable(False)
self._scroll.setStyleSheet(f"QScrollArea{{border:none;background:{_BG};}}")
self._tl = _TimelineWidget()
self._tl.row_clicked.connect(self._on_row_clicked)
self._tl.prog_clicked.connect(self._show_prog_detail)
self._scroll.setWidget(self._tl)
vl.addWidget(self._scroll, stretch=1)

t = QTimer(self)
t.timeout.connect(self._tick)
t.start(1000)
self._tick()

def _tick(self):
    self._clock.setText(datetime.now().strftime("%I:%M %p    %b %d, %Y"))

def _on_back(self):
    self.player.stop()
```



```

self.back_clicked.emit()

def load(self, streams: list, cat_name: str, scroll_to_now: bool = True):
    self._scroll_to_now = scroll_to_now
    self._current_streams = streams
    self._current_cat = cat_name
    self._day_offset = 0
    self._day_lbl.setText("Today")
    self._load_day(streams, cat_name)

def _load_day(self, streams: list, cat_name: str):
    if len(streams) > MAX_EPG_CHANNELS:
        streams = streams[:MAX_EPG_CHANNELS]
        self._cat_lbl.setText(f"{cat_name} (first {MAX_EPG_CHANNELS})")
    else:
        self._cat_lbl.setText(cat_name)

    today = datetime.now().replace(hour=0, minute=0, second=0, microsecond=0)
    from datetime import timedelta
    start = today + timedelta(days=self._day_offset)
    self._start = start
    self._rows = [{"name": s.get("name", "-"), "programs": [], "stream": s} for s in streams]
    self._tl.set_data(self._rows, start)

    # scroll to start of selected day
    if self._day_offset == 0 and self._scroll_to_now:
        now_px = int((datetime.now() - start).total_seconds() / 60 * MINS_W)
        QTimer.singleShot(60, lambda: self._scroll.horizontalScrollBar().setValue(
            max(0, CH_W + now_px - 300)))
    else:
        QTimer.singleShot(60, lambda: self._scroll.horizontalScrollBar().setValue(0))

    self._epg_gen += 1
    self._epg_queue = list(enumerate(streams))
    self._epg_active = 0
    self._drain_epg()

    self._icon_gen += 1
    self._icon_queue = [(i, s) for i, s in enumerate(streams)
                        if s.get("stream_icon")]
    self._icon_active = 0
    self._flush_icons(self._icon_gen)

def _drain_epg(self):
    while self._epg_queue and self._epg_active < 8:
        idx, s = self._epg_queue.pop(0)
        self._epg_active += 1
        sid = s.get("stream_id")
        gen = self._epg_gen
        w = Worker(lambda i=sid: self._api.get_short_epg(i, limit=10))
        w.signals.result.connect(lambda d, i=idx, g=gen: self._got_epg(i, d, g))
        w.signals.error.connect(lambda _: self._epg_err())
        QThreadPool.globalInstance().start(w)

def _got_epg(self, idx, data, gen):
    self._epg_active -= 1
    if gen == self._epg_gen and idx < len(self._rows):
        self._rows[idx]["programs"] = data.get("epg_listings", [])
        self._tl.update()
    self._drain_epg()

```

```
def _epg_err(self):
    self._epg_active -= 1
    self._drain_epg()

def _flush_icons(self, gen: int):
    sz = self._tl._AV_SZ
    while self._icon_queue and self._icon_active < 4:
        idx, s = self._icon_queue.pop(0)
        self._icon_active += 1
        url = s.get("stream_icon", "")
        w = Worker(lambda u=url: fetch_logo_bytes(u))
        w.signals.result.connect(
            lambda data, i=idx, g=gen: self._on_icon_done(i, data, g))
        w.signals.error.connect(lambda _, g=gen: self._on_icon_err(g))
        QThreadPool.globalInstance().start(w)

def _on_icon_done(self, idx: int, data: bytes, gen: int):
    self._icon_active -= 1
    if gen == self._icon_gen and data:
        px = QPixmap()
        if px.loadFromData(data):
            self._tl.update_avatar(idx, round_pixmap(px, self._tl._AV_SZ))
    self._flush_icons(gen)

def _on_icon_err(self, gen: int):
    self._icon_active -= 1
    self._flush_icons(gen)

def _shift_day(self, delta: int):
    if not self._current_streams:
        return
    self._day_offset += delta
    from datetime import timedelta
    today = datetime.now().replace(hour=0, minute=0, second=0, microsecond=0)
    target = today + timedelta(days=self._day_offset)
    if self._day_offset == 0:
        self._day_lbl.setText("Today")
    elif self._day_offset == 1:
        self._day_lbl.setText("Tomorrow")
    elif self._day_offset == -1:
        self._day_lbl.setText("Yesterday")
    else:
        self._day_lbl.setText(target.strftime("%b %d"))
    # Use higher EPG limit for future days so we get enough entries
    epg_limit = min(10 + abs(self._day_offset) * 40, 200)
    self._load_day(self._current_streams, self._current_cat)
    self._epg_gen += 1
    streams = self._current_streams[:MAX_EPG_CHANNELS]
    self._epg_queue = [(i, s, epg_limit) for i, s in enumerate(streams)]
    self._epg_active = 0
    self._drain_epg_with_limit(self._epg_gen)

def _drain_epg_with_limit(self, gen: int):
    while self._epg_queue and self._epg_active < 8:
        idx, s, limit = self._epg_queue.pop(0)
        self._epg_active += 1
        sid = s.get("stream_id")
        w = Worker(lambda i=sid, lim=limit: self._api.get_short_epg(i, limit=lim))
        w.signals.result.connect(lambda d, i=idx, g=gen: self._got_epg(i, d, g))
```

```
w.signals.error.connect(lambda _: self._epg_err())
QThreadPool.globalInstance().start(w)

def _on_row_clicked(self, row: int):
    self._tl.select_row(row)
    ch = self._rows[row]
    s = ch["stream"]
    url = self._api.live_url(s.get("stream_id"))

    now = datetime.now()
    now_prog = None
    for prog in ch.get("programs", []):
        try:
            ps = datetime.fromtimestamp(int(prog.get("start_timestamp", 0)))
            pe = datetime.fromtimestamp(int(prog.get("stop_timestamp", 0)))
            if ps <= now <= pe:
                now_prog = prog
                break
        except Exception:
            pass

    if now_prog:
        title = _decode(now_prog.get("title", "")) or "No Information"
        ts = datetime.fromtimestamp(int(now_prog["start_timestamp"])).strftime("%I:%M %p")
        te = datetime.fromtimestamp(int(now_prog["stop_timestamp"])).strftime("%I:%M %p")
        self._prog_title.setText(title)
        self._prog_time.setText(f"{ts} - {te}")
    else:
        self._prog_title.setText(s.get("name", ""))
        self._prog_time.setText("")

    self._current_url = url
    self.player.play(url)
    self.channel_activated.emit(s, url)

def _show_prog_detail(self, prog: dict):
    from PyQt6.QtWidgets import QApplication
    title = prog.get("title") or "No Information"
    desc = prog.get("description") or "No description available."
    try:
        ts = datetime.fromtimestamp(int(prog["start_timestamp"])).strftime("%I:%M %p")
        te = datetime.fromtimestamp(int(prog["stop_timestamp"])).strftime("%I:%M %p")
        time_str = f"{ts} - {te}"
    except Exception:
        time_str = ""

    dlg = QDialog(self)
    dlg.setWindowFlags(Qt.WindowType.Popup | Qt.WindowType.FramelessWindowHint)
    dlg.setStyleSheet(
        f"QDialog{{background:#0d1b2a;border:1px solid #1e2a3a;border-radius:8px;}}"
        f"QLabel{{background:transparent;}}"
    )
    vl = QVBoxLayout(dlg)
    vl.setContentsMargins(16, 14, 16, 14)
    vl.setSpacing(6)

    t_lbl = QLabel(title)
    tf = QFont(); tf.setPixelSize(14); tf.setBold(True)
    t_lbl.setFont(tf)
    t_lbl.setStyleSheet("color:#ffffff;")
```

```
t_lbl.setWordWrap(True)
vl.addWidget(t_lbl)

if time_str:
    tm_lbl = QLabel(time_str)
    tm_lbl.setStyleSheet("color:#42a5f5; font-size:11px;")
    vl.addWidget(tm_lbl)

if desc and desc != "No description available.":
    from PyQt6.QtWidgets import QFrame
    sep = QFrame(); sep.setFrameShape(QFrame.Shape.HLine)
    sep.setStyleSheet("color:#1e2a3a;")
    vl.addWidget(sep)
    d_lbl = QLabel(desc)
    d_lbl.setStyleSheet("color:#d0d8e8; font-size:12px;")
    d_lbl.setWordWrap(True)
    d_lbl.setMaximumWidth(340)
    vl.addWidget(d_lbl)

cursor_pos = QApplication.instance().primaryScreen().geometry()
from PyQt6.QtGui import QCursor
dlg.move(QCursor.pos())
dlg.adjustSize()
QTimer.singleShot(8000, dlg.close)
dlg.exec()

def select_and_play(self, idx: int):
    if 0 <= idx < len(self._rows):
        self._on_row_clicked(idx)

def _open_fullscreen(self):
    if not self._current_url:
        return
    self.player.stop()
    dlg = _FullscreenPlayer(self._current_url, self)
    dlg.exec()
    # Resume in the small player after returning
    self.player.play(self._current_url)

# — top-level EPG container —————

class EPGBScreen(QWidget):
    back_to_home = pyqtSignal()
    channel_activated = pyqtSignal(dict, str)

    def __init__(self, stream_filter=None, cat_title: str = "EPG Categories", parent=None):
        super().__init__(parent)
        self._api = None
        self._config = None
        self._stream_filter = stream_filter
        vl = QVBoxLayout(self)
        vl.setContentsMargins(0, 0, 0, 0)
        self._stack = QStackedWidget()
        vl.addWidget(self._stack)

        self._cats = EPGBCategoryScreen(title=cat_title)
        self._cats.back_clicked.connect(self.back_to_home)
        self._cats.category_selected.connect(self._open_grid)
        self._stack.addWidget(self._cats) # index 0
```

```
self._grid: EPGridScreen | None = None # created once api is known

def set_api(self, api):
    self._api = api
    if self._grid is not None:
        self._grid._api = api

def set_config(self, config):
    self._config = config

def open(self):
    self._stack.setCurrentIndex(0)
    if self._api is None:
        return
    w = Worker(self._api.get_live_categories)
    w.signals.result.connect(self._on_cats_loaded)
    w.signals.error.connect(lambda _: None)
    QThreadPool.globalInstance().start(w)

def load_direct(self, streams: list, name: str):
    """Skip category picker – go straight to the grid with the given streams."""
    if self._grid is None:
        self._grid = EPGridScreen(self._api)
        self._grid.back_clicked.connect(lambda: self.back_to_home.emit())
        self._grid.channel_activated.connect(self.channel_activated)
        self._stack.addWidget(self._grid) # index 1
    self._stack.setCurrentIndex(1)
    self._grid.load(streams, name, scroll_to_now=False)

def _on_cats_loaded(self, cats: list):
    if self._config:
        saved = self._config.cat_order.get("0", [])
        if saved:
            order_map = {str(cid): i for i, cid in enumerate(saved)}
            cats = sorted(cats, key=lambda c: order_map.get(
                str(c.get("category_id", "")), len(saved)))
        specials = []
        if self._config.favorite_streams:
            specials.append({"category_name": " FAVORITES",
                            "category_id": "__FAVORITES__", "_color": "#f9a825"})
        if self._config.recent_streams:
            specials.append({"category_name": " RECENT",
                            "category_id": "__RECENT__", "_color": "#42a5f5"})
        cats = specials + cats
    self._cats.load_categories(cats)

def _open_grid(self, cat_id, name):
    if self._api is None:
        return
    # Create EPGridScreen lazily here so PlayerWidget's native X11 window
    # is created while EPGridScreen is already visible in the main stack.
    if self._grid is None:
        self._grid = EPGridScreen(self._api)
        self._grid.back_clicked.connect(lambda: self._stack.setCurrentIndex(0))
        self._grid.channel_activated.connect(self.channel_activated)
        self._stack.addWidget(self._grid) # index 1
    self._stack.setCurrentIndex(1)
    def _apply_filter(streams):
        if self._stream_filter:
```

```

        return [s for s in streams if self._stream_filter(s)]
    return streams

    if cat_id == "__FAVORITES__" and self._config:
        self._grid.load(_apply_filter(self._config.favorite_streams), name)
    elif cat_id == "__RECENT__" and self._config:
        self._grid.load(_apply_filter(self._config.recent_streams), name)
    else:
        w = Worker(lambda: self._api.get_live_streams(cat_id))
        w.signals.result.connect(lambda streams: self._grid.load(_apply_filter(streams), name))
        w.signals.error.connect(lambda _: None)
        QThreadPool.globalInstance().start(w)

# — Radio station picker —————

class RadioScreen(QWidget):
    """Category-style 3-column grid of radio stations. Click → play immediately."""
    back_to_home      = pyqtSignal()
    channel_activated = pyqtSignal(dict, str)
    stop_requested    = pyqtSignal()

    _ROW_H = 34 # shorter than default _CatRow (48px) so more stations fit

    _PAGE = 500

    _MAX_PROBE = 16 # concurrent probe workers

    def __init__(self, parent=None):
        super().__init__(parent)
        self._api      = None
        self._config    = None
        self._streams: list = []
        self._all_streams: list = []
        self._offset     = 0
        self._fav_ids: set = set()
        self._recent_ids: list = [] # ordered most-recent first
        self._favs_only  = False
        self._recent_only = False
        self._hidden_only = False
        self._hidden_streams: list = [] # radio streams in hidden_stream_ids
        # dead-stream probing
        self._probed_ids: set = set() # all sids probed this session (alive or dead)
        self._probe_queue: list = [] # (sid, url, gen) tuples waiting to probe
        self._probe_active: int = 0
        self._probe_gen: int = 0 # incremented per page render to cancel stale results
        self._hide_timer = QTimer(self) # debounce re-render after hiding streams
        self._hide_timer.setSingleShot(True)
        self._hide_timer.setInterval(800)
        self._hide_timer.timeout.connect(self._render_page)
        self.setStyleSheet(f"background:{_BG};")

        vl = QVBoxLayout(self)
        vl.setContentsMargins(0, 0, 0, 0)
        vl.setSpacing(0)

        hdr = QWidget()
        hdr.setFixedHeight(52)
        hdr.setStyleSheet(f"background:{_PANEL};border-bottom:1px solid {_BORDER};")
        hl = QHBoxLayout(hdr)

```

```
hl.setContentsMargins(16, 0, 16, 0)
hl.setSpacing(12)
back = _hdr_btn("-")
back.clicked.connect(self.back_to_home)
hl.addWidget(back)
lbl = QLabel("Radio Stations")
lbl.setStyleSheet("color:white;font-size:15px;font-weight:bold;background:transparent;")
hl.addWidget(lbl, stretch=1)
self._fav_btn = QPushButton(" FAVORITES")
self._fav_btn.setCheckable(True)
self._fav_btn.setCursor(Qt.CursorShape.PointingHandCursor)
self._fav_btn.setStyleSheet(
    "QPushButton{background:#1e2a3a;color:#4a6a8a;border:none;"
    "border-radius:6px;font-size:12px;font-weight:bold;padding:6px 12px;}"
    "QPushButton:checked{background:#2a3a1a;color:#f9a825;}"
    "QPushButton:hover{color:white;}"
)
self._fav_btn.toggled.connect(self._toggle_favs)
hl.addWidget(self._fav_btn)
self._recent_btn = QPushButton(" RECENT")
self._recent_btn.setCheckable(True)
self._recent_btn.setCursor(Qt.CursorShape.PointingHandCursor)
self._recent_btn.setStyleSheet(
    "QPushButton{background:#1e2a3a;color:#4a6a8a;border:none;"
    "border-radius:6px;font-size:12px;font-weight:bold;padding:6px 12px;}"
    "QPushButton:checked{background:#1a2a3a;color:#42a5f5;}"
    "QPushButton:hover{color:white;}"
)
self._recent_btn.toggled.connect(self._toggle_recent)
hl.addWidget(self._recent_btn)
self._hidden_btn = QPushButton(" HIDDEN")
self._hidden_btn.setCheckable(True)
self._hidden_btn.setCursor(Qt.CursorShape.PointingHandCursor)
self._hidden_btn.setStyleSheet(
    "QPushButton{background:#1e2a3a;color:#4a6a8a;border:none;"
    "border-radius:6px;font-size:12px;font-weight:bold;padding:6px 12px;}"
    "QPushButton:checked{background:#2a1a1a;color:#c62828;}"
    "QPushButton:hover{color:white;}"
)
self._hidden_btn.toggled.connect(self._toggle_hidden)
hl.addWidget(self._hidden_btn)
stop = QPushButton("")
stop.setFixedSize(44, 44)
stop.setToolTip("Stop")
stop.setCursor(Qt.CursorShape.PointingHandCursor)
stop.setStyleSheet(
    "QPushButton{background:transparent;color:#42a5f5;border:none;font-size:28px;}"
    "QPushButton:hover{color:white;}"
)
stop.clicked.connect(self.stop_requested)
hl.addWidget(stop)
vl.addWidget(hdr)

scroll = QScrollArea()
scroll.setWidgetResizable(True)
scroll.setStyleSheet("QScrollArea{border:none;background:transparent;}")
self._inner = QWidget()
self._inner.setStyleSheet(f"background:{_BG};")
self._station_grid = QGridLayout(self._inner)
self._station_grid.setContentsMargins(16, 12, 16, 12)
```

```

        self._station_grid.setSpacing(5)
        for _c in range(4):
            self._station_grid.setColumnStretch(_c, 1)
        scroll.setWidget(self._inner)
        self._scroll = scroll
        vl.addWidget(scroll, stretch=1)

# — pagination footer —————
self._footer = QWidget()
self._footer.setFixedHeight(44)
self._footer.setStyleSheet(
    f"background:{_PANEL};border-top:1px solid {_BORDER};"
)
fl = QHBoxLayout(self._footer)
fl.setContentsMargins(16, 0, 16, 0)
fl.setSpacing(12)
self._prev_btn = self._nav_btn("← Prev")
self._prev_btn.clicked.connect(self._prev_page)
fl.addWidget(self._prev_btn)
self._page_lbl = QLabel("")
self._page_lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
self._page_lbl.setStyleSheet("color:#aaa;font-size:12px;background:transparent;")
fl.addWidget(self._page_lbl, stretch=1)
self._next_btn = self._nav_btn("Next 500 →")
self._next_btn.clicked.connect(self._next_page)
fl.addWidget(self._next_btn)
self._footer.hide()
vl.addWidget(self._footer)

@staticmethod
def _nav_btn(text: str) -> QPushButton:
    b = QPushButton(text)
    b.setCursor(Qt.CursorShape.PointingHandCursor)
    b.setStyleSheet(
        "QPushButton{background:#1e2a3a;color:#42a5f5;border:none;"
        "border-radius:6px;font-size:13px;font-weight:bold;padding:6px 14px;}"
        "QPushButton:hover{background:#253545;color:white;}"
    )
    return b

def set_api(self, api):
    self._api = api

def set_config(self, config):
    self._config = config

@staticmethod
def _valid_sid(s: dict) -> bool:
    """Return True only when stream_id is a positive integer (Xstream standard).
    Filters out corrupt provider entries whose stream_id contains HTML or garbage."""
    try:
        return int(s.get("stream_id", 0)) > 0
    except (ValueError, TypeError):
        return False

def load(self, streams: list, name: str):
    if self._config:
        self._fav_ids = {str(x) for x in self._config.radio_favorite_ids}
        self._recent_ids = [str(s.get("stream_id", "")) for s in self._config.radio_recent_streams]
        fav = self._fav_ids

```



```

hidden = ({str(x) for x in self._config.hidden_stream_ids}
          if self._config else set())
# build full index including extras from favourites/recents
all_by_id = {str(s.get("stream_id", "")): s for s in streams if self._valid_sid(s)}
if self._config:
    for s in self._config.radio_favorite_streams + self._config.radio_recent_streams:
        sid = str(s.get("stream_id", ""))
        if sid and sid not in all_by_id and self._valid_sid(s):
            all_by_id[sid] = s
self._all_streams = sorted(
    [s for s in all_by_id.values() if str(s.get("stream_id", "")) not in hidden],
    key=lambda s: (0 if str(s.get("stream_id", "")) in fav else 1)
)
self._hidden_streams = [s for s in all_by_id.values()
                        if str(s.get("stream_id", "")) in hidden]
self._update_hidden_btn_label()
self._offset = 0
self._favs_only = False
self._recent_only = False
self._hidden_only = False
for btn in (self._fav_btn, self._recent_btn, self._hidden_btn):
    btn.blockSignals(True)
    btn.setChecked(False)
    btn.blockSignals(False)
self._render_page()

def _render_page(self):
    if self._hidden_only:
        self._render_hidden_page()
        return
    if self._recent_only:
        recent_order = {sid: i for i, sid in enumerate(self._recent_ids)}
        source = sorted(
            [s for s in self._all_streams if str(s.get("stream_id", "")) in recent_order],
            key=lambda s: recent_order.get(str(s.get("stream_id", "")), 999)
        )
    elif self._favs_only:
        source = [s for s in self._all_streams if str(s.get("stream_id", "")) in self._fav_ids]
    else:
        source = self._all_streams
    self._streams = source[self._offset:self._offset + self._PAGE]

    # Remove old cells synchronously: reparent each to a temporary absorber
    # widget, then let Python's refcount destroy it immediately. This avoids
    # the zombie-wrapper crash that comes from calling setWidget() again.
    _absorber = QWidget()
    while self._station_grid.count():
        item = self._station_grid.takeAt(0)
        w = item.widget()
        if w:
            w.setParent(_absorber)
    del _absorber # immediate C++ deletion – no lingering old buttons

    # cancel any in-flight probes from the previous page
    self._probe_gen += 1
    self._probe_queue = []
    self._probe_active = 0
    gen = self._probe_gen

    fav = self._fav_ids

```

```

for idx, s in enumerate(self._streams):
    sid = str(s.get("stream_id", ""))
    is_fav = sid in fav

    cell = QWidget()
    hl = QHBoxLayout(cell)
    hl.setContentsMargins(0, 0, 0, 0)
    hl.setSpacing(2)

    play = QPushButton("▶ " + s.get("name", "-"))
    play.setFixedHeight(34)
    play.setCursor(Qt.CursorShape.PointingHandCursor)
    play.setProperty("station_idx", idx)
    self._style_live_btn(play)
    play.clicked.connect(self._on_play_clicked)
    hl.addWidget(play, stretch=1)

    star = QPushButton("★" if is_fav else "☆")
    star.setFixedSize(28, 34)
    star.setCursor(Qt.CursorShape.PointingHandCursor)
    star.setProperty("station_idx", idx)
    self._style_star(star, is_fav)
    star.clicked.connect(self._on_star_clicked)
    hl.addWidget(star)

    r, c = divmod(idx, 4)
    self._station_grid.addWidget(cell, r, c)

# queue probes for all stations on this page (skip already-probed ones)
if self._api:
    hidden = ({str(x) for x in self._config.hidden_stream_ids}
              if self._config else set())
    for s in self._streams:
        sid = str(s.get("stream_id", ""))
        if sid and sid not in self._probed_ids:
            url = self._api.live_url(s.get("stream_id"))
            self._probe_queue.append((sid, url, gen))
    self._flush_probe_queue()

remainder = len(self._streams) % 4
if remainder:
    last_row = len(self._streams) // 4
    for col in range(remainder, 4):
        self._station_grid.addWidget(QWidget(), last_row, col)

# update footer
if self._recent_only:
    total = len([s for s in self._all_streams if str(s.get("stream_id", "")) in set(self._recent_ids)])
elif self._favs_only:
    total = len([s for s in self._all_streams if str(s.get("stream_id", "")) in self._fav_ids])
else:
    total = len(self._all_streams)
start = self._offset + 1
end = min(self._offset + self._PAGE, total)
self._page_lbl.setText(f"Stations {start}-{end} of {total}")
self._prev_btn.setVisible(self._offset > 0)
self._next_btn.setVisible(end < total)
self._footer.setVisible(total > self._PAGE)
self._scroll.verticalScrollBar().setValue(0)

```

```

@staticmethod
def _style_star(btn: QPushButton, is_fav: bool):
    color = "#f9a825" if is_fav else "#4a6a8a"
    btn.setStyleSheet(
        f"QPushButton{{background:#162230;color:{color};border:none;font-size:15px;"
        "border-top-right-radius:6px;border-bottom-right-radius:6px;}}"
        "QPushButton:hover{color:#f9a825;}")
    )

@staticmethod
def _style_live_btn(btn: QPushButton):
    btn.setStyleSheet(
        f"QPushButton{{background:#162230;color:{_TEXT};font-size:12px;"
        "font-weight:bold;border:none;"
        "border-top-left-radius:6px;border-bottom-left-radius:6px;"
        "text-align:left;padding:0 8px;}}"
        "QPushButton:hover{background:#1a2d40;}")
    )

def _toggle_hidden(self, checked: bool):
    if checked:
        self._fav_btn.blockSignals(True); self._fav_btn.setChecked(False); self._fav_btn.blockSignals(False)
        self._recent_btn.blockSignals(True); self._recent_btn.setChecked(False); self._recent_btn.blockSignals(False)
        self._favs_only = False
        self._recent_only = False
    self._hidden_only = checked
    self._offset = 0
    self._render_page()

def _update_hidden_btn_label(self):
    n = len(self._hidden_streams)
    self._hidden_btn.setText(f"  HIDDEN ({n})" if n else "  HIDDEN")

def _render_hidden_page(self):
    """Render the hidden-streams page with per-station restore buttons."""
    _absorber = QWidget()
    while self._station_grid.count():
        item = self._station_grid.takeAt(0)
        w = item.widget()
        if w:
            w.setParent(_absorber)
    del _absorber

    source = self._hidden_streams[self._offset:self._offset + self._PAGE]

    for idx, s in enumerate(source):
        cell = QWidget()
        hl = QHBoxLayout(cell)
        hl.setContentsMargins(0, 0, 0, 0)
        hl.setSpacing(2)

        name_btn = QPushButton(s.get("name", "-"))
        name_btn.setFixedHeight(34)
        name_btn.setEnabled(False)
        name_btn.setStyleSheet(
            "QPushButton{background:#1e1212;color:#7a4a4a;font-size:12px;"
            "font-weight:bold;border:none;"
            "border-top-left-radius:6px;border-bottom-left-radius:6px;"
            "text-align:left;padding:0 8px;}")
    )

```

```

hl.addWidget(name_btn, stretch=1)

restore = QPushButton("↶")
restore.setFixedSize(36, 34)
restore.setToolTip("Restore station")
restore.setCursor(Qt.CursorShape.PointingHandCursor)
restore.setProperty("station_sid", str(s.get("stream_id", "")))
restore.setStyleSheet(
    "QPushButton{background:#1a2a1a;color:#4caf50;border:none;font-size:14px;"
    "border-top-right-radius:6px;border-bottom-right-radius:6px;}"
    "QPushButton:hover{background:#1e3a1e;color:#81c784;}"
)
restore.clicked.connect(self._on_restore_clicked)
hl.addWidget(restore)

r, c = divmod(idx, 4)
self._station_grid.addWidget(cell, r, c)

remainder = len(source) % 4
if remainder:
    for col in range(remainder, 4):
        self._station_grid.addWidget(QWidget(), len(source) // 4, col)

total = len(self._hidden_streams)
start, end = self._offset + 1, min(self._offset + self._PAGE, total)
self._page_lbl.setText(f"Hidden stations {start}--{end} of {total}" if total else "No hidden stations")
self._prev_btn.setVisible(self._offset > 0)
self._next_btn.setVisible(end < total)
self._footer.setVisible(total > self._PAGE)
self._scroll.verticalScrollBar().setValue(0)

def _on_restore_clicked(self):
    btn = self.sender()
    if not btn or not self._config:
        return
    sid = btn.property("station_sid")
    if not sid:
        return
    # Remove from hidden list
    self._config.hidden_stream_ids = [
        x for x in self._config.hidden_stream_ids if str(x) != sid
    ]
    self._config.save()
    # Move from hidden to visible
    restored = [s for s in self._hidden_streams if str(s.get("stream_id", "")) == sid]
    self._hidden_streams = [s for s in self._hidden_streams if str(s.get("stream_id", "")) != sid]
    self._all_streams = sorted(
        self._all_streams + restored,
        key=lambda s: (0 if str(s.get("stream_id", "")) in self._fav_ids else 1)
    )
    # Allow re-probing this station
    self._probed_ids.discard(sid)
    self._update_hidden_btn_label()
    self._render_page()

# — stream health probing —————

def _flush_probe_queue(self):
    while self._probe_queue and self._probe_active < self._MAX_PROBE:
        sid, url, gen = self._probe_queue.pop(0)

```

```

        self._probe_active += 1
        w = Worker(self._do_probe, url)
        w.signals.result.connect(
            lambda ok, s=sid, g=gen: self._on_probe_result(s, ok, g)
        )
        w.signals.error.connect(
            lambda _e, s=sid, g=gen: self._on_probe_result(s, False, g)
        )
        QThreadPool.globalInstance().start(w)

    @staticmethod
    def _do_probe(url: str) -> bool:
        import requests as _req
        from urllib.parse import urljoin

        import sys as _sys
        _UA = ("Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 Chrome/120.0.0.0 Safari/537.36"
              if _sys.platform == "win32" else
              "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 Chrome/120.0.0.0 Safari/537.36")
        _AUDIO_CT = (
            'audio/', 'video/mp2t', 'video/mpeg',
            'application/ogg', 'application/x-ogg',
            'application/vnd.apple.mpegurl', 'application/x-mpegurl',
        )
        _MAGIC = (
            b'\xff\xfb', b'\xff\xf3', b'\xff\xf2', # MP3
            b'\xff\xf1', b'\xff\xf9', # AAC ADTS
            b'ID3', # MP3 with ID3 tag
            b'OggS', # Ogg (Vorbis/Opus)
            b'fLaC', # FLAC
            b'RIFX', # WAV
            b'\x47', # MPEG-TS sync byte
        )

    def _fetch(target, use_range=True):
        hdrs = {"User-Agent": _UA}
        if use_range:
            hdrs["Range"] = "bytes=0-4095"
        try:
            r = _req.get(target, timeout=4, stream=True, headers=hdrs)
            if r.status_code >= 400:
                return None, '', b''
            ct = r.headers.get('content-type', '').lower().split(';')[0].strip()
            body = b''
            for data in r.iter_content(chunk_size=4096):
                body += data
                if len(body) >= 4096:
                    break
            return r.status_code, ct, body
        except Exception:
            return None, '', b''

    def _is_hls(body):
        return b'#EXTM3U' in body[:64] or b'#EXT-X-' in body[:64]

    def _first_segment(body, base):
        for line in body.decode('utf-8', errors='ignore').splitlines():
            line = line.strip()
            if line and not line.startswith('#'):
                return line if line.startswith('http') else urljoin(base, line)

```

```
        return None

    def _is_audio(ct, body):
        if len(body) < 64:
            return False
        for prefix in _AUDIO_CT:
            if ct.startswith(prefix):
                return True
        for magic in _MAGIC:
            if body[:len(magic)] == magic:
                return True
        return False

    try:
        # Don't use Range for .m3u8 so we get the full manifest text
        status, ct, body = _fetch(url, use_range=not url.endswith('.m3u8'))
        if status is None:
            return False

        # Follow HLS chain up to 2 hops (master playlist → media playlist → segment)
        for _ in range(2):
            if not _is_hls(body):
                break
            seg = _first_segment(body, url)
            if not seg:
                return False
            url = seg
            status, ct, body = _fetch(url)
            if status is None:
                return False

        return _is_audio(ct, body)
    except Exception:
        return False

def _on_probe_result(self, sid: str, alive: bool, gen: int):
    self._probe_active -= 1
    self._probed_ids.add(sid)
    if gen == self._probe_gen and not alive and self._config:
        hidden_set = {str(x) for x in self._config.hidden_stream_ids}
        if sid not in hidden_set:
            self._config.hidden_stream_ids.append(sid)
            self._config.save()
        # Move from visible to hidden in memory
        moved = [s for s in self._all_streams if str(s.get("stream_id", "")) == sid]
        self._all_streams = [s for s in self._all_streams if str(s.get("stream_id", "")) != sid]
        self._hidden_streams = self._hidden_streams + moved
        self._update_hidden_btn_label()
        self._hide_timer.start()
    self._flush_probe_queue()

def _on_play_clicked(self, checked=False):
    btn = self.sender()
    if btn is None:
        return
    self._on_station_selected(int(btn.property("station_idx")))

def _on_star_clicked(self, checked=False):
    star = self.sender()
    if star is None:
```

```
        return

    idx = int(star.property("station_idx"))
    if idx >= len(self._streams):
        return

    sid = str(self._streams[idx].get("stream_id", ""))
    is_fav = sid not in self._fav_ids # toggle
    if is_fav:
        self._fav_ids.add(sid)
    else:
        self._fav_ids.discard(sid)
    star.setText("★" if is_fav else "☆")
    self._style_star(star, is_fav)
    if self._config:
        self._config.radio_favorite_ids = list(self._fav_ids)
        s = self._streams[idx]
        sid_val = s.get("stream_id")
        self._config.radio_favorite_streams = [
            x for x in self._config.radio_favorite_streams
            if str(x.get("stream_id", "")) != str(sid_val)
        ]
        if is_fav:
            self._config.radio_favorite_streams.append(s)
        self._config.save()

def _toggle_favs(self, checked: bool):
    if checked:
        self._recent_btn.blockSignals(True)
        self._recent_btn.setChecked(False)
        self._recent_btn.blockSignals(False)
        self._recent_only = False
    self._favs_only = checked
    self._offset = 0
    self._render_page()

def _toggle_recent(self, checked: bool):
    if checked:
        self._fav_btn.blockSignals(True)
        self._fav_btn.setChecked(False)
        self._fav_btn.blockSignals(False)
        self._favs_only = False
    self._recent_only = checked
    self._offset = 0
    self._render_page()

def _next_page(self):
    self._offset += self._PAGE
    QTimer.singleShot(0, self._render_page)

def _prev_page(self):
    self._offset = max(0, self._offset - self._PAGE)
    QTimer.singleShot(0, self._render_page)

def _on_station_selected(self, idx: int):
    if self._api is None or idx >= len(self._streams):
        return
    s = self._streams[idx]
    sid = s.get("stream_id")
    if not self._valid_sid(s):
        from .app_logger import AppLogger
        AppLogger.get().error()
```

```

        f"RadioScreen: skipping unplayable stream_id={sid!r} name={s.get('name')!r}"
    )
    return
if self._config:
    sid_str = str(sid)
    self._config.radio_recent_streams = [
        x for x in self._config.radio_recent_streams
        if str(x.get("stream_id", "")) != sid_str
    ]
    self._config.radio_recent_streams.insert(0, s)
    self._config.radio_recent_streams = self._config.radio_recent_streams[:20]
    self._recent_ids = [str(x.get("stream_id", "")) for x in self._config.radio_recent_streams]
    self._config.save()
url = self._api.live_url(sid)
self.channel_activated.emit(s, url)

```

lxplayer/multi_screen.py

```

from PyQt6.QtWidgets import (
    QWidget, QVBoxLayout, QHBoxLayout, QLabel, QPushButton,
    QGridLayout, QScrollArea, QStackedWidget, QListWidget, QListWidgetItem,
    QLineEdit,
)
from PyQt6.QtCore import Qt, QTimer, pyqtSignal, QThreadPool, QSize
from PyQt6.QtGui import QPainter, QColor, QPainterPath, QIcon, QPixmap

from .player import PlayerWidget
from .workers import Worker
from .utils import make_avatar, fetch_logo_bytes, round_pixmap
from .delegates import ChannelDelegate, EPG_ROLE, EPG_PROG_ROLE

_BG      = "#0a0f1a"
_PANEL   = "#0d1117"
_BORDER  = "#1e2a3a"
_TEXT    = "#d0d8e8"
_DIM     = "#4a6a8a"
_SEL     = "#1565c0"

GRID_R = GRID_C = 6  # all layouts proportioned on a 6x6 grid

# Each layout: list of (row, col, rowspan, colspan)
LAYOUTS = [
    [(0,0,3,3),(0,3,3,3),(3,0,3,3),(3,3,3,3)],      # 2x2
    [(0,0,6,4),(0,4,2,2),(2,4,2,2),(4,4,2,2)],      # 1 large + 3 stacked
    [(0,0,3,4),(0,4,3,2),(3,0,3,3),(3,3,3,3)],      # 1 large + 2 right + 2 bottom
    [(0,0,3,6),(3,0,3,3),(3,3,3,3)],                 # 1 top + 2 bottom
    [(0,0,6,3),(0,3,6,3)],                           # 2 equal
    [(0,0,4,6),(4,0,2,6)],                           # 1 large + 1 small
]

def _hdr_btn(text: str) -> QPushButton:
    b = QPushButton(text)
    b.setFixedSize(32, 32)
    b.setStyleSheet(
        "QPushButton{background:transparent;color:#42a5f5;border:none;font-size:18px;}"
        "QPushButton:hover{color:white;}"
    )

```



```
    return b

# — multil: layout picker —————

class _LayoutCard(QWidget):
    clicked = pyqtSignal(int)

    def __init__(self, idx: int, panels: list, parent=None):
        super().__init__(parent)
        self._idx = idx
        self._panels = panels
        self._hover = False
        self.setCursor(Qt.CursorShape.PointingHandCursor)
        self.setFixedSize(200, 130)

    def paintEvent(self, _):
        p = QPainter(self)
        p.setRenderHint(QPainter.RenderHint.Antialiasing)
        W, H = float(self.width()), float(self.height())
        PAD = 14.0
        outer = QPainterPath()
        outer.addRoundedRect(2.0, 2.0, W - 4.0, H - 4.0, 10.0, 10.0)
        p.fillPath(outer, QColor("#1e2e42" if self._hover else "#1a2535"))
        p.setPen(QColor("#42a5f5" if self._hover else "#2a3a4a"))
        p.drawPath(outer)

        iw = W - 2.0 * PAD
        ih = H - 2.0 * PAD
        for row, col, rs, cs in self._panels:
            bx = PAD + col / GRID_C * iw + 2.0
            by = PAD + row / GRID_R * ih + 2.0
            bw = cs / GRID_C * iw - 4.0
            bh = rs / GRID_R * ih - 4.0
            pp = QPainterPath()
            pp.addRoundedRect(bx, by, bw, bh, 3.0, 3.0)
            p.fillPath(pp, QColor("#0a1520"))
            p.setPen(QColor("#2a5a8a" if self._hover else "#1e3a5a"))
            p.drawPath(pp)

    def enterEvent(self, _): self._hover = True; self.update()
    def leaveEvent(self, _): self._hover = False; self.update()
    def mousePressEvent(self, e):
        if e.button() == Qt.MouseButton.LeftButton:
            self.clicked.emit(self._idx)

class MultiLayoutPicker(QWidget):
    layout_selected = pyqtSignal(int)
    back_clicked = pyqtSignal()

    def __init__(self, parent=None):
        super().__init__(parent)
        self.setStyleSheet(f"background:{_BG};")
        vl = QVBoxLayout(self)
        vl.setContentsMargins(0, 0, 0, 0)
        vl.setSpacing(0)

        hdr = QWidget()
        hdr.setFixedHeight(52)
```

```

        hdr.setStyleSheet(f"background:{_PANEL};border-bottom:1px solid {_BORDER};")
        hl = QHBoxLayout(hdr)
        hl.setContentsMargins(16, 0, 16, 0)
        hl.setSpacing(12)
        back = _hdr_btn("-")
        back.clicked.connect(self.back_clicked)
        hl.addWidget(back)
        title = QLabel("Choose Multi-Screen Layout")
        title.setStyleSheet("color:white;font-size:15px;font-weight:bold;background:transparent;")
        hl.addWidget(title, stretch=1)
        vl.addWidget(hdr)

        inner = QWidget()
        inner.setStyleSheet(f"background:{_BG};")
        grid = QGridLayout(inner)
        grid.setContentsMargins(48, 48, 48, 48)
        grid.setSpacing(32)
        for i, panels in enumerate(LAYOUTS):
            row, col = divmod(i, 3)
            card = _LayoutCard(i, panels)
            card.clicked.connect(self.layout_selected)
            grid.addWidget(card, row, col, Qt.AlignmentFlag.AlignCenter)

        scroll = QScrollArea()
        scroll.setWidgetResizable(True)
        scroll.setStyleSheet("QScrollArea{border:none;background:transparent;}")
        scroll.setWidget(inner)
        vl.addWidget(scroll, stretch=1)

# — multi2: category picker —————

class MultiCategoryPicker(QWidget):
    category_selected = pyqtSignal(object, str)  # (cat_id or None, name)
    back_clicked      = pyqtSignal()

    def __init__(self, parent=None):
        super().__init__(parent)
        self.setStyleSheet(f"background:{_BG};")
        vl = QVBoxLayout(self)
        vl.setContentsMargins(0, 0, 0, 0)
        vl.setSpacing(0)

        hdr = QWidget()
        hdr.setFixedHeight(52)
        hdr.setStyleSheet(f"background:{_PANEL};border-bottom:1px solid {_BORDER};")
        hl = QHBoxLayout(hdr)
        hl.setContentsMargins(16, 0, 16, 0)
        hl.setSpacing(12)
        back = _hdr_btn("-")
        back.clicked.connect(self.back_clicked)
        hl.addWidget(back)
        self._hdr_lbl = QLabel("Select Category")
        self._hdr_lbl.setStyleSheet(
            "color:white;font-size:15px;font-weight:bold;background:transparent;")
        hl.addWidget(self._hdr_lbl, stretch=1)
        vl.addWidget(hdr)

        self._list = QListWidget()
        self._list.setStyleSheet(f"")

```

```

        QListWidget{{background:{_PANEL};border:none;color:{_TEXT};font-size:13px;}}
        QListWidget::item{{padding:12px 18px;border-bottom:1px solid {_BORDER};}}
        QListWidget::item:selected{{background:{_SEL};color:white;}}
        QListWidget::item:hover:!selected{{background:#151e2a;}}
    """)
    self._list.itemClicked.connect(self._on_item)
    vl.addWidget(self._list, stretch=1)

def set_label(self, text: str):
    self._hdr_lbl.setText(text)

def load(self, cats: list):
    self._list.clear()
    _SPECIAL = ("__FAVORITES__", "__RECENT__")
    # pinned specials first
    for c in cats:
        if c.get("category_id") in _SPECIAL:
            item = QListWidgetItem(f" {c.get('category_name', '-')}")
            item.setData(Qt.ItemDataRole.UserRole,
                        (c.get("category_id"), c.get("category_name", "-")))
            color = c.get("_color")
            if color:
                item.setForeground(QColor(color))
            self._list.addItem(item)

    # ALL
    all_item = QListWidgetItem(" ALL")
    all_item.setData(Qt.ItemDataRole.UserRole, (None, "ALL"))
    self._list.addItem(all_item)
    # regular categories
    for c in cats:
        if c.get("category_id") not in _SPECIAL:
            item = QListWidgetItem(f" {c.get('category_name', '-')}")
            item.setData(Qt.ItemDataRole.UserRole,
                        (c.get("category_id"), c.get("category_name", "-")))
            self._list.addItem(item)

def _on_item(self, item: QListWidgetItem):
    cid, name = item.data(Qt.ItemDataRole.UserRole)
    self.category_selected.emit(cid, name)

# — multi3: channel picker —————

class MultiChannelPicker(QWidget):
    channel_selected = pyqtSignal(dict, str) # stream, url
    back_clicked = pyqtSignal()

    def __init__(self, parent=None):
        super().__init__(parent)
        self.setStyleSheet(f"background:{_BG};")
        self._api = None
        self._streams = []
        self._icon_gen = 0
        self._icon_active = 0
        self._icon_queue: list = []
        self._epg_gen = 0
        self._epg_active = 0
        self._epg_queue: list = []

        vl = QVBoxLayout(self)

```

```
vl.setContentsMargins(0, 0, 0, 0)
vl.setSpacing(0)

hdr = QWidget()
hdr.setFixedHeight(52)
hdr.setStyleSheet(f"background:{_PANEL};border-bottom:1px solid {_BORDER};")
hl = QHBoxLayout(hdr)
hl.setContentsMargins(16, 0, 16, 0)
hl.setSpacing(12)
back = _hdr_btn("-")
back.clicked.connect(self.back_clicked)
hl.addWidget(back)
self._hdr_lbl = QLabel("Select Channel")
self._hdr_lbl.setStyleSheet(
    "color:white;font-size:15px;font-weight:bold;background:transparent;"
)
hl.addWidget(self._hdr_lbl, stretch=1)
vl.addWidget(hdr)

sw = QWidget()
sw.setFixedHeight(44)
sw.setStyleSheet(f"background:{_PANEL};border-bottom:1px solid {_BORDER};")
sl = QHBoxLayout(sw)
sl.setContentsMargins(12, 5, 12, 5)
self._search = QLineEdit()
self._search.setPlaceholderText("Search channels...")
self._search.setStyleSheet(f"""
    QLineEdit{{background:#111827;border:1px solid {_BORDER};
        border-radius:5px;padding:6px 10px;color:{_TEXT};font-size:12px;}}
    QLineEdit:focus{{border-color:#1976d2;}}
""")
self._search.textChanged.connect(self._filter)
sl.addWidget(self._search)
vl.addWidget(sw)

self._list = QListWidget()
self._list.setStyleSheet(f"""
    QListWidget{{background:{_PANEL};border:none;color:{_TEXT};font-size:13px;}}
    QListWidget::item{{border-bottom:1px solid {_BORDER};}}
    QListWidget::item:selected{{background:{_SEL};color:white;}}
    QListWidget::item:hover:!selected{{background:#151e2a;}}
""")
self._list.setIconSize(QSize(36, 36))
self._delegate = ChannelDelegate()
self._list.setItemDelegate(self._delegate)
self._list.itemClicked.connect(self._on_item)
vl.addWidget(self._list, stretch=1)

def set_header(self, text: str):
    self._hdr_lbl.setText(text)

def load(self, streams: list, api):
    self._api = api
    self._streams = streams
    self._search.clear()
    self._list.clear()
    for s in streams:
        name = s.get("name", "-")
        item = QListWidgetItem(name)
        item.setData(Qt.ItemDataRole.UserRole, s)
        item.setData(EPG_ROLE, "")
```

```
        item.setIcon(QIcon(make_avatar(name, 36)))
        self._list.addItem(item)

    self._icon_gen += 1
    self._icon_queue = [(i, s) for i, s in enumerate(streams)
                        if s.get("stream_icon")]
    self._icon_active = 0
    self._flush_icons(self._icon_gen)

    self._epg_gen += 1
    self._epg_queue = [(i, s) for i, s in enumerate(streams)
                      if s.get("stream_id")]
    self._epg_active = 0
    self._flush_epg(self._epg_gen)

def _flush_icons(self, gen: int):
    while self._icon_queue and self._icon_active < 4:
        idx, s = self._icon_queue.pop(0)
        self._icon_active += 1
        url = s.get("stream_icon", "")
        w = Worker(lambda u=url: fetch_logo_bytes(u))
        w.signals.result.connect(
            lambda data, i=idx, g=gen: self._on_icon_done(i, data, g))
        w.signals.error.connect(lambda _, g=gen: self._on_icon_err(g))
        QThreadPool.globalInstance().start(w)

def _on_icon_done(self, idx: int, data: bytes, gen: int):
    self._icon_active -= 1
    if gen == self._icon_gen and data:
        px = QPixmap()
        if px.loadFromData(data):
            item = self._list.item(idx)
            if item:
                item.setIcon(QIcon(round_pixmap(px, 32)))
    self._flush_icons(gen)

def _on_icon_err(self, gen: int):
    self._icon_active -= 1
    self._flush_icons(gen)

def _flush_epg(self, gen: int):
    while self._epg_queue and self._epg_active < 8:
        idx, s = self._epg_queue.pop(0)
        self._epg_active += 1
        sid = s.get("stream_id")
        w = Worker(lambda i=sid: self._api.get_short_epg(i, limit=20))
        w.signals.result.connect(
            lambda data, i=idx, g=gen: self._on_epg_done(i, data, g))
        w.signals.error.connect(lambda _, g=gen: self._on_epg_err(g))
        QThreadPool.globalInstance().start(w)

def _on_epg_done(self, idx: int, data: dict, gen: int):
    self._epg_active -= 1
    if gen == self._epg_gen:
        listings = data.get("epg_listings", [])
        if listings:
            from datetime import datetime as _dt
            now_dt = _dt.now()
            current = None
            for listing in listings:
```

```
        try:
            s = int(listing.get("start_timestamp") or 0)
            t = int(listing.get("stop_timestamp") or 0)
            if s and t and _dt.fromtimestamp(s) <= now_dt <= _dt.fromtimestamp(t):
                current = listing
                break
        except Exception:
            pass
    e = current or listings[0]
    item = self._list.item(idx)
    if item:
        import base64
        raw = e.get("title", "")
        try:
            title = base64.b64decode(raw + "==").decode("utf-8") if raw else ""
        except Exception:
            title = raw
        if title:
            item.setData(EPG_ROLE, title)
        try:
            s = int(e.get("start_timestamp") or 0)
            t = int(e.get("stop_timestamp") or 0)
            if s and t:
                item.setData(EPG_PROG_ROLE, (s, t))
        except Exception:
            pass
    self._flush_epg(gen)

def _on_epg_err(self, gen: int):
    self._epg_active -= 1
    self._flush_epg(gen)

def _filter(self, text: str):
    for i in range(self._list.count()):
        item = self._list.item(i)
        s = item.data(Qt.ItemDataRole.UserRole)
        name = s.get("name", "") if s else item.text()
        item.setHidden(bool(text) and text.lower() not in name.lower())

def _on_item(self, item: QListWidgetItem):
    s = item.data(Qt.ItemDataRole.UserRole)
    if s and self._api:
        self.channel_selected.emit(s, self._api.live_url(s.get("stream_id")))

# — fullscreen overlay —————

# — Panel slot (placeholder → live player) —————

class _PanelSlot(QWidget):
    assign_requested = pyqtSignal()
    panel_clicked = pyqtSignal(str) # url

    def __init__(self, slot_num: int, parent=None):
        super().__init__(parent)
        self._url = ""
        self.player: PlayerWidget | None = None
```

```

vl = QVBoxLayout(self)
vl.setContentsMargins(0, 0, 0, 0)
vl.setSpacing(0)

# — panel header bar (hidden until a channel is assigned) —————
self._bar = QWidget()
self._bar.setFixedHeight(24)
self._bar.setStyleSheet("background:#050a10;border-bottom:1px solid #0d1a26;")
bl = QHBoxLayout(self._bar)
bl.setContentsMargins(6, 0, 4, 0)
bl.setSpacing(4)
self._bar_lbl = QLabel(f"Panel {slot_num + 1}")
self._bar_lbl.setStyleSheet("color:#4a6a8a;font-size:10px;background:transparent;")
bl.addWidget(self._bar_lbl, stretch=1)
self._fs_btn = QPushButton("")
self._fs_btn.setFixedSize(20, 20)
self._fs_btn.setToolTip("Fullscreen")
self._fs_btn.setStyleSheet(
    "QPushButton{background:transparent;color:#42a5f5;border:none;font-size:13px;}"
    "QPushButton:hover{color:white;}"
)
self._fs_btn.clicked.connect(self._on_fs_clicked)
bl.addWidget(self._fs_btn)
self._bar.hide()
vl.addWidget(self._bar)

self._inner = QStackedWidget()
vl.addWidget(self._inner, stretch=1)

ph = QWidget()
ph.setStyleSheet("background:#050a10;")
ph.setCursor(Qt.CursorShape.PointingHandCursor)
pl = QVBoxLayout(ph)
lbl = QLabel(f"+\nPanel {slot_num + 1}")
lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
lbl.setStyleSheet(
    f"color:{_DIM};font-size:22px;font-weight:bold;background:transparent;"
)
pl.addWidget(lbl)
ph.mousePressEvent = lambda e: (
    self.assign_requested.emit()
    if e.button() == Qt.MouseButton.LeftButton else None
)
self._inner.addWidget(ph) # index 0 — placeholder

def _on_fs_clicked(self):
    if self._url:
        self.panel_clicked.emit(self._url)

def assign(self, url: str, name: str = ""):
    self._url = url
    if self.player is None:
        self.player = PlayerWidget()
        self.player.setMinimumSize(0, 0) # allow shrinking in grid
        self._inner.addWidget(self.player) # index 1
        self.player.clicked.connect(lambda: self.panel_clicked.emit(self._url))
    if name:
        self._bar_lbl.setText(name)
    self._bar.show()
    self._inner.setCurrentIndex(1)
    self.player.play(url)

```

```

def stop(self):
    if self.player:
        self.player.stop()

# — multi4: multi-screen view —————

class MultiScreenView(QWidget):
    back_clicked = pyqtSignal()
    slot_fill_needed = pyqtSignal(int) # emitted when user clicks "+" on a panel

    def __init__(self, parent=None):
        super().__init__(parent)
        self.setStyleSheet(f"background:{_BG};")
        self._slots: list[_PanelSlot] = []
        self.layout_idx = -1
        self._fs_slot: _PanelSlot | None = None
        self._fs_url: str = ""
        self._pre_fs_state = None # saved window state before fullscreen

        vl = QVBoxLayout(self)
        vl.setContentsMargins(0, 0, 0, 0)
        vl.setSpacing(0)

        hdr = QWidget()
        hdr.setFixedHeight(44)
        hdr.setStyleSheet(f"background:{_PANEL};border-bottom:1px solid {_BORDER};")
        hl = QHBoxLayout(hdr)
        hl.setContentsMargins(12, 0, 12, 0)
        back = _hdr_btn("←")
        back.clicked.connect(self.back_clicked)
        hl.addWidget(back)
        self._title = QLabel("Multi-Screen")
        self._title.setStyleSheet(
            "color:white;font-size:13px;font-weight:bold;background:transparent;"
        )
        hl.addWidget(self._title, stretch=1)
        self._mute_btn = QPushButton(" Mute All")
        self._mute_btn.setStyleSheet(
            f"QPushButton{{background:transparent;color:{_DIM};border:none;font-size:12px;}}"
            f"QPushButton:hover{{color:white;}}"
        )
        self._muted = False
        self._mute_btn.clicked.connect(self._toggle_mute)
        hl.addWidget(self._mute_btn)
        self._hdr = hdr
        vl.addWidget(hdr)

        # — body: grid (page 0) ↔ single fullscreen player (page 1) —————
        self._body = QStackedWidget()
        vl.addWidget(self._body, stretch=1)

        # page 0 — panel grid
        self._grid_w = QWidget()
        self._grid_w.setStyleSheet(f"background:{_BG};")
        self._grid_l = QGridLayout(self._grid_w)
        self._grid_l.setContentsMargins(4, 4, 4, 4)
        self._grid_l.setSpacing(4)
        self._body.addWidget(self._grid_w) # index 0

```



```

# page 1 - fullscreen single player
fs_w = QWidget()
fs_w.setStyleSheet("background:black;")
fs_vl = QVBoxLayout(fs_w)
fs_vl.setContentsMargins(0, 0, 0, 0)
fs_vl.setSpacing(0)

fs_bar = QWidget()
fs_bar.setFixedHeight(40)
fs_bar.setStyleSheet(f"background:{_PANEL};border-bottom:1px solid {_BORDER};")
fs_hl = QHBoxLayout(fs_bar)
fs_hl.setContentsMargins(12, 0, 12, 0)
fs_back = QPushButton("← Back to Multi-Screen")
fs_back.setStyleSheet(
    "QPushButton{background:transparent;color:#42a5f5;border:none;font-size:13px;}"
    "QPushButton:hover{color:white;}"
)
fs_back.clicked.connect(self._close_fullscreen)
fs_hl.addWidget(fs_back)
fs_hl.addStretch()
fs_hint = QLabel("ESC · click video")
fs_hint.setStyleSheet("color:#4a6a8a;font-size:11px;background:transparent;")
fs_hl.addWidget(fs_hint)
fs_vl.addWidget(fs_bar)

self._fs_player = PlayerWidget()
self._fs_player.clicked.connect(self._close_fullscreen)
fs_vl.addWidget(self._fs_player, stretch=1)

self._body.addWidget(fs_w)          # index 1

def setup(self, layout_idx: int):
    self.layout_idx = layout_idx
    self._fs_player.stop()
    self._body.setCurrentIndex(0)
    for slot in self._slots:
        slot.stop()
        self._grid_l.removeWidget(slot)
        slot.deleteLater()
    self._slots.clear()

    for i in range(GRID_R):
        self._grid_l.setRowStretch(i, 1)
    for i in range(GRID_C):
        self._grid_l.setColumnStretch(i, 1)

    for i, (row, col, rs, cs) in enumerate(LAYOUTS[layout_idx]):
        slot = _PanelSlot(i)
        slot.assign_requested.connect(
            lambda checked=False, idx=i: self.slot_fill_needed.emit(idx))
        slot.panel_clicked.connect(
            lambda url, s=slot: self._open_fullscreen(url, s))
        self._slots.append(slot)
        self._grid_l.addWidget(slot, row, col, rs, cs)

    self._title.setText(f"Multi-Screen · {len(self._slots)} panels")

def assign(self, slot_idx: int, url: str, name: str = ""):
    if 0 <= slot_idx < len(self._slots):
        self._slots[slot_idx].assign(url, name)

```

```
def stop_all(self):
    for s in self._slots:
        s.stop()
    self._fs_player.stop()
    self._body.setCurrentIndex(0)

def _toggle_mute(self):
    self._muted = not self._muted
    vol = 0 if self._muted else 80
    for slot in self._slots:
        if slot.player:
            slot.player.set_volume(vol)
    self._mute_btn.setText(" Unmute All" if self._muted else " Mute All")

def _open_fullscreen(self, url: str, slot: _PanelSlot):
    if slot.player:
        slot.player.stop()
    self._fs_slot = slot
    self._fs_url = url

    # Save window state then go fullscreen
    top = self.window()
    self._pre_fs_state = top.windowState()
    self._hdr.hide()
    self._body.setCurrentIndex(1)
    top.showFullScreen()

    QTimer.singleShot(80, lambda: self._fs_player.play(url))

def _close_fullscreen(self):
    self._fs_player.stop()
    self._hdr.show()
    self._body.setCurrentIndex(0)

    # Restore window state
    top = self.window()
    state = self._pre_fs_state
    self._pre_fs_state = None
    if state is not None:
        top.setWindowState(state)
    else:
        top.showNormal()

    slot, url = self._fs_slot, self._fs_url
    self._fs_slot = None
    self._fs_url = ""
    if slot and slot.player and url:
        QTimer.singleShot(80, lambda: slot.player.play(url))

def keyPressEvent(self, e):
    if e.key() == Qt.Key.Key_Escape and self._body.currentIndex() == 1:
        self._close_fullscreen()
    else:
        super().keyPressEvent(e)
```

```
# — top-level MultiScreen container —————
```

```
class MultiScreen(QWidget):
```

```
back_to_home = pyqtSignal()

def __init__(self, parent=None):
    super().__init__(parent)
    self._api = None
    self._config = None
    self._cats: list = []
    self._filling = 0

    vl = QVBoxLayout(self)
    vl.setContentsMargins(0, 0, 0, 0)
    self._stack = QStackedWidget()
    vl.addWidget(self._stack)

    self._picker = MultiLayoutPicker()
    self._picker.layout_selected.connect(self._on_layout)
    self._picker.back_clicked.connect(self.back_to_home)
    self._stack.addWidget(self._picker) # 0

    self._cat = MultiCategoryPicker()
    self._cat.category_selected.connect(self._on_cat)
    self._cat.back_clicked.connect(self._cat_back)
    self._stack.addWidget(self._cat) # 1

    self._ch = MultiChannelPicker()
    self._ch.channel_selected.connect(self._on_channel)
    self._ch.back_clicked.connect(lambda: self._stack.setCurrentIndex(1))
    self._stack.addWidget(self._ch) # 2

    self._view = MultiScreenView()
    self._view.back_clicked.connect(self._on_view_back)
    self._view.slot_fill_needed.connect(self._fill_slot)
    self._stack.addWidget(self._view) # 3

def set_api(self, api):
    self._api = api

def set_config(self, config):
    self._config = config

def open(self):
    self._stack.setCurrentIndex(0)

# — internal flow —————

def _on_layout(self, idx: int):
    self._view.setup(idx)
    # Open channel picker for the first panel immediately
    self._fill_slot(0)

def _fill_slot(self, slot_idx: int):
    self._filling = slot_idx
    n = len(self._view._slots)
    self._cat.set_label(f"Select Category – Panel {slot_idx + 1} of {n}")
    self._stack.setCurrentIndex(1)
    if self._cats:
        self._cat.load(self._cats)
    elif self._api:
        w = Worker(self._api.get_live_categories)
        w.signals.result.connect(self._on_cats_loaded)
```

```

w.signals.error.connect(lambda _: None)
QThreadPool.globalInstance().start(w)

def _on_cats_loaded(self, cats: list):
    if self._config:
        saved = self._config.cat_order.get("0", [])
        if saved:
            order_map = {str(cid): i for i, cid in enumerate(saved)}
            cats = sorted(cats, key=lambda c: order_map.get(
                str(c.get("category_id", "")), len(saved)))
        specials = []
        if self._config.favorite_streams:
            specials.append({"category_name": " FAVORITES",
                            "category_id": "__FAVORITES__", "_color": "#f9a825"})
        if self._config.recent_streams:
            specials.append({"category_name": " RECENT",
                            "category_id": "__RECENT__", "_color": "#42a5f5"})
        cats = specials + cats
    self._cats = cats
    self._cat.load(cats)

    # If no favorites saved, auto-navigate to the Canadian category
    if self._config and not self._config.favorite_streams:
        canadian = next(
            (c for c in cats if "canada" in str(c.get("category_name", "")).lower()),
            None,
        )
        if canadian:
            QTimer.singleShot(0, lambda: self._on_cat(
                canadian["category_id"], canadian["category_name"]
            ))

def _on_cat(self, cat_id, cat_name: str):
    n = len(self._view._slots)
    self._ch.set_header(f"{cat_name} - Panel {self._filling + 1} of {n}")
    self._stack.setCurrentIndex(2)
    if cat_id == "__FAVORITES__" and self._config:
        self._ch.load(self._config.favorite_streams, self._api)
    elif cat_id == "__RECENT__" and self._config:
        self._ch.load(self._config.recent_streams, self._api)
    elif self._api:
        w = Worker(lambda: self._api.get_live_streams(cat_id))
        w.signals.result.connect(lambda s: self._ch.load(s, self._api))
        w.signals.error.connect(lambda _: None)
        QThreadPool.globalInstance().start(w)

def _on_channel(self, stream, url: str):
    slot = self._filling
    name = stream.get("name", "") if isinstance(stream, dict) else ""
    # Show the grid BEFORE creating the PlayerWidget so its native window
    # is initialised inside a visible parent hierarchy.
    self._stack.setCurrentIndex(3)
    QTimer.singleShot(60, lambda: self._view.assign(slot, url, name))

def _cat_back(self):
    # Go back to the grid if one is already set up, otherwise to picker
    if self._view.layout_idx >= 0:
        self._stack.setCurrentIndex(3)
    else:
        self._stack.setCurrentIndex(0)

```

```

def _on_view_back(self):
    self._view.stop_all()
    self._view.layout_idx = -1
    self._cats = []
    self._stack.setCurrentIndex(0)

```

lxplayer/main_window.py

```

import base64
import os
import sys
import time
import requests as _requests
from datetime import datetime

from PyQt6.QtWidgets import (
    QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,
    QListWidget, QListWidgetItem, QLabel, QLineEdit,
    QPushButton, QSlider, QFrame, QMessageBox,
    QStackedWidget, QSplitter, QDialog, QMenu,
    QScrollArea, QStyledItemDelegate, QStyle,
)
from PyQt6.QtCore import Qt, QThreadPool, QTimer, QSize, QPoint, pyqtSignal
from PyQt6.QtGui import (
    QFont, QKeySequence, QShortcut, QIcon, QPixmap, QImage,
    QColor, QPainter, QLinearGradient, QPen,
)

from .config import Config, Account, QUALITY_CACHE_FILE, VOD_QUALITY_CACHE_FILE, SEASON_CACHE_FILE, POSTER_CACHE_DIR
from .xstream_api import XstreamAPI
from .workers import Worker
from .player import PlayerWidget
from .login_dialog import LoginDialog
from .app_logger import AppLogger
from .log_panel import LogPanel
from .delegates import ChannelDelegate, PosterDelegate, EPG_ROLE, EPG_PROG_ROLE, RADIO_ROLE, FAV_ROLE, QUALITY_ROLE, FAIL_>
    ROLE
from .home_screen import HomeScreen
from .epg_screen import EPGSscreen, RadioScreen
from .multi_screen import MultiScreen
from .utils import make_avatar as _make_avatar, make_radio_avatar as _make_radio_avatar, is_radio_stream as _is_radio_stre>
    am

# — colours —————
_BG          = "#0a0f1a"
_PANEL       = "#0d1117"
_ITEM_BG     = "#111827"
_SEL         = "#1565c0"
_BORDER      = "#1e2a3a"
_TEXT        = "#d0d8e8"
_DIM         = "#4a6a8a"

_STYLE = f"""
QWidget {{ background: {_BG}; color: {_TEXT}; }}
QWidget#cat_panel {{ background: {_PANEL}; border-right: 1px solid {_BORDER}; }}
QWidget#ch_panel  {{ background: {_PANEL}; border-right: 1px solid {_BORDER}; }}

```

```
QWidget#topbar    {{ background: {_BG}; border-bottom: 1px solid {_BORDER}; }}
QWidget#controls  {{ background: transparent; border-top: 1px solid {_BORDER}; }}
QWidget#epg_bar   {{ background: transparent; border-top: 1px solid {_BORDER}; }}
QWidget#back_bar  {{ background: transparent; border-bottom: 1px solid {_BORDER}; }}
QListWidget {{
    background: {_PANEL}; border: none; outline: none;
    font-size: 13px; color: {_TEXT};
}}
QListWidget::item {{ border-bottom: 1px solid #151e2a; }}
QListWidget::item:selected {{
    background: #0d2044;
    color: #fff;
    border: 1px solid #42a5f5;
    border-radius: 3px;
}}
QListWidget::item: hover:!selected {{ background: #151e2a; color: #fff; }}
QListWidget#cat_list::item {{
    padding: 9px 14px; font-size: 12px; color: {_DIM};
}}
QListWidget#cat_list::item:selected {{
    background: #0d2044; color: #42a5f5;
    border-left: 3px solid #42a5f5;
    border-top: 1px solid #1976d2;
    border-bottom: 1px solid #1976d2;
    border-right: 1px solid #1976d2;
}}
QListWidget#grid_list {{
    background: {_BG};
}}
QListWidget#grid_list::item {{
    padding: 3px; border: none; color: transparent;
}}
QListWidget#grid_list::item:selected {{
    background: #0d204488;
    border: 2px solid #42a5f5;
    border-radius: 4px;
}}
QLineEdit {{
    background: #111827; border: 1px solid {_BORDER};
    border-radius: 5px; padding: 7px 11px; color: {_TEXT}; font-size: 12px;
}}
QLineEdit:focus {{ border-color: #1976d2; }}
QPushButton#icon_btn {{
    background: transparent; color: {_DIM}; border: none;
    border-radius: 4px; font-size: 16px; padding: 4px 8px;
}}
QPushButton#icon_btn: hover {{ color: #fff; background: #1a2535; }}
QPushButton#tab_btn {{
    background: transparent; color: {_DIM}; border: none;
    border-bottom: 2px solid transparent;
    padding: 6px 16px; font-size: 12px; font-weight: bold;
}}
QPushButton#tab_btn: checked {{ color: #fff; border-bottom: 2px solid #1976d2; }}
QPushButton#tab_btn: hover: !checked {{ color: #aac; }}
QPushButton#login_btn {{
    background: #1976d2; color: white; border: none; border-radius: 4px;
    font-size: 11px; font-weight: bold; padding: 5px 10px;
}}
QPushButton#login_btn: hover {{ background: #1e88e5; }}
QPushButton#back_btn {{
```

```

        background: transparent; color: #42a5f5; border: none;
        font-size: 13px; padding: 4px 12px;
    }}
QPushButton#back_btn:hover {{ color: #fff; }}
QSlider::groove:horizontal {{
    height: 4px; background: #1e2a3a; border-radius: 2px;
}}
QSlider::handle:horizontal {{
    background: #1976d2; width: 14px; height: 14px;
    margin: -5px 0; border-radius: 7px;
}}
QSlider::sub-page:horizontal {{ background: #1976d2; border-radius: 2px; }}
QSlider#seek_bar::groove:horizontal {{
    height: 5px; background: #1e2a3a; border-radius: 2px;
}}
QSlider#seek_bar::handle:horizontal {{
    background: #42a5f5; width: 12px; height: 12px;
    margin: -4px 0; border-radius: 6px;
}}
QSlider#seek_bar::sub-page:horizontal {{ background: #42a5f5; border-radius: 2px; }}
QScrollBar:vertical {{ background: transparent; width: 5px; }}
QScrollBar::handle:vertical {{ background: #2a3a4a; border-radius: 2px; min-height: 20px; }}
QScrollBar::add-line:vertical, QScrollBar::sub-line:vertical {{ height: 0; }}
QSplitter::handle {{ background: {_BORDER}; }}
"""

_POSTER_W, _POSTER_H = 120, 180
_LOGO_SZ = 36

def _quality_from_name(name: str) -> str | None:
    n = name.upper()
    if "4K" in n or "UHD" in n or "2160" in n:
        return "4K"
    if "FHD" in n or "1080" in n:
        return "1080p"
    if "HD" in n or "720" in n:
        return "720p"
    if "SD" in n or "480" in n or "360" in n:
        return "SD"
    return None

import re as _re
_PREFIX_RE = _re.compile(
    r'^[A-Z]{2,3}[\s]*[:\-\-][\s]*|'
    r'^(?:UK|US|CA|FR|DE|AU|NL|BE|IT|ES|PT|IE|IN|PK|TR|AR|MX|BR|RU|PL|SE|NO|DK|FI|NZ|HK|SG|ZA|AE|SA|EG|KR|JP|CN|TW|MY|TH|P>
H|GR|CY|IL|HR|HU|RO|BG|CZ|UA|TN|MA|DZ|IQ|IR|QA|KW|JO|LB|BD|LK|NP|AF|VN|KH|ET|KE|GH|NG|TZ|UG|CM|AO|MZ|ZW|RW|SD|SO|GY|VE>
|CO|EC|PE|BO|PY|UY|CL|PA|CR|NI|HN|GT|SV|DO|HT|JM|CU|BB|TT|PR)\s+',
    _re.IGNORECASE,
)

def _clean_name(raw: str) -> str:
    return _PREFIX_RE.sub("", raw, count=1).strip()

def _stream_quality_hint(s: dict, cached: str | None = None) -> str | None:
    """Return the best quality label we can determine without probing the stream.
    Returns None when quality is genuinely unknown (no badge shown)."""
    if cached:

```

```

        return cached
    name = s.get("name", "")
    q = _quality_from_name(name)
    if q:
        return q
    # Check is_hd / is_uhd flags many Xstream providers include
    if s.get("is_uhd") or str(s.get("is_uhd", "")).lower() in ("1", "true"):
        return "4K"
    if s.get("is_hd") or str(s.get("is_hd", "")).lower() in ("1", "true"):
        return "720p"
    return "SD" # default - M3U8 probe will upgrade if resolution is higher

def _quality_color(label: str) -> str:
    """Return badge background colour for a quality label."""
    return {
        "4K": "#6a1b9a", # purple
        "1080p": "#1565c0", # blue
        "720p": "#00695c", # teal
        "SD": "#4a4a4a", # grey
    }.get(label, "#2a3a4a") # dark fallback

BASE_POSTER_ROLE = Qt.ItemDataRole.UserRole + 6
VOD_QUALITY_ROLE = Qt.ItemDataRole.UserRole + 7
WATCH_PROG_ROLE = Qt.ItemDataRole.UserRole + 8

def _stream_quality_label(w: int, h: int) -> str:
    """Map pixel dimensions to a quality badge label.
    Checks both dimensions so widescreen movies (e.g. 1280x534) get the right tier."""
    if w >= 3840 or h >= 2160: return "4K"
    if w >= 1920 or h >= 1080: return "1080p"
    if w >= 1280 or h >= 720: return "720p"
    if w >= 854 or h >= 480: return "SD"
    return "SD"

def _fav_id(s: dict) -> str:
    return str(s.get("stream_id") or s.get("series_id") or "")

class _MiniPlayer(QDialog):
    """Small floating picture-in-picture player, draggable by its title bar."""
    closed = pyqtSignal()

    def __init__(self, url: str, title: str, parent=None):
        super().__init__(parent)
        self._url = url
        self._drag_pos: QPoint | None = None
        self.setWindowFlags(
            Qt.WindowType.Window |
            Qt.WindowType.FramelessWindowHint |
            Qt.WindowType.WindowStaysOnTopHint
        )
        self.setFixedSize(360, 220)

        vl = QVBoxLayout(self)
        vl.setContentsMargins(0, 0, 0, 0)
        vl.setSpacing(0)

```



```

        bar = QWidget()
        bar.setFixedHeight(28)
        bar.setStyleSheet("background:#0d1117;")
        hl = QHBoxLayout(bar)
        hl.setContentsMargins(8, 0, 4, 0)
        lbl = QLabel(title[:50])
        lbl.setStyleSheet("color:#d0d8e8;font-size:11px;background:transparent;")
        hl.addWidget(lbl, stretch=1)
        x_btn = QPushButton("x")
        x_btn.setFixedSize(22, 22)
        x_btn.setStyleSheet(
            "QPushButton{background:transparent;color:#4a6a8a;border:none;font-size:11px;}"
            "QPushButton:hover{color:white;background:#c62828;border-radius:3px;}"
        )
        x_btn.clicked.connect(self.close)
        hl.addWidget(x_btn)
        vl.addWidget(bar)

        self.player = PlayerWidget()
        self.player.setMinimumSize(0, 0)
        self.player.clicked.connect(self.close) # click video → return to main
        vl.addWidget(self.player, stretch=1)

        QTimer.singleShot(80, lambda: self.player.play(url))

    def mousePressEvent(self, e):
        if e.button() == Qt.MouseButton.LeftButton:
            self._drag_pos = e.globalPosition().toPoint() - self.frameGeometry().topLeft()

    def mouseMoveEvent(self, e):
        if self._drag_pos and e.buttons() & Qt.MouseButton.LeftButton:
            self.move(e.globalPosition().toPoint() - self._drag_pos)

    def mouseReleaseEvent(self, e):
        self._drag_pos = None

    def closeEvent(self, e):
        self.player.stop()
        self.closed.emit()
        super().closeEvent(e)

class _GlobalSearchDialog(QDialog):
    """Search across all live channels regardless of category."""
    stream_selected = pyqtSignal(dict, str)

    def __init__(self, api, parent=None):
        super().__init__(parent)
        self._api = api
        self._streams: list = []
        self.setWindowTitle("Search All Channels")
        self.setMinimumSize(480, 520)
        self.setModal(True)
        self._build()
        self._search_edit.setPlaceholderText("Loading all channels...")
        self._search_edit.setEnabled(False)
        w = Worker(lambda: api.get_live_streams())
        w.signals.result.connect(self._on_loaded)
        w.signals.error.connect(lambda _: self._search_edit.setPlaceholderText("Load failed"))

```

```
        QThreadPool.globalInstance().start(w)

def _build(self):
    vl = QVBoxLayout(self)
    vl.setSpacing(8)
    vl.setContentsMargins(14, 14, 14, 14)

    self._search_edit = QLineEdit()
    self._search_edit.setPlaceholderText("Search channels...")
    self._search_edit.textChanged.connect(self._filter)
    vl.addWidget(self._search_edit)

    self._list = QListWidget()
    self._list.setStyleSheet(
        f"QListWidget{{background:#0d1117;border:none;color:#d0d8e8;font-size:13px;}}"
        f"QListWidget::item{{padding:8px 14px;border-bottom:1px solid #1e2a3a;}}"
        f"QListWidget::item:selected{{background:#1565c0;color:white;}}"
        f"QListWidget::item:hover:!selected{{background:#151e2a;}}"
    )
    self._list.itemDoubleClicked.connect(self._on_item)
    self._list.itemClicked.connect(self._on_item)
    vl.addWidget(self._list, stretch=1)

    self._count_lbl = QLabel("")
    self._count_lbl.setStyleSheet("color:#4a6a8a;font-size:11px;")
    vl.addWidget(self._count_lbl)

def _on_loaded(self, streams: list):
    self._streams = streams
    self._search_edit.setEnabled(True)
    self._search_edit.setPlaceholderText("Search all channels...")
    self._search_edit.setFocus()
    self._filter("")

def _filter(self, text: str):
    self._list.clear()
    q = text.strip().lower()
    shown = 0
    for s in self._streams:
        name = s.get("name", "")
        if not q or q in name.lower():
            item = QListWidgetItem(name)
            item.setData(Qt.ItemDataRole.UserRole, s)
            self._list.addItem(item)
            shown += 1
            if shown >= 300:
                break
    total = len(self._streams)
    self._count_lbl.setText(
        f"Showing {shown} of {total}" if q else f"{total} channels total"
    )

def _on_item(self, item: QListWidgetItem):
    s = item.data(Qt.ItemDataRole.UserRole)
    if s:
        url = self._api.live_url(s.get("stream_id"))
        self.stream_selected.emit(s, url)
        self.accept()
```

```
class _CatList(QListWidget):
    """Category list that prevents items being dropped above the pinned header rows."""

    def __init__(self, parent=None):
        super().__init__(parent)
        self.pinned_rows = 0

    def dropEvent(self, event):
        if self.pinned_rows > 0:
            last_pin = self.item(self.pinned_rows - 1)
            if last_pin:
                if event.position().y() <= self.visualItemRect(last_pin).bottom():
                    event.ignore()
                    return
            super().dropEvent(event)

class _VodPopup(QDialog):
    """Full-window dim-overlay dialog – Play (default/Enter) or Favorites. Click outside card to dismiss."""

    def __init__(self, s: dict, item, is_fav: bool, quality: str, parent=None, synopsis: str = "", show_plot: bool = True):
        super().__init__(parent)
        self.setWindowFlags(Qt.WindowType.FramelessWindowHint | Qt.WindowType.Dialog)
        self.setAttribute(Qt.WidgetAttribute.WA_TranslucentBackground)
        self.setWindowModality(Qt.WindowModality.WindowModal)

        if parent is not None:
            self.setGeometry(parent.window().geometry())

        self.play_clicked = False
        self.fav_toggled = False
        self._is_fav = is_fav

        self.setStyleSheet("""
            QDialog { background: transparent; }
            QWidget#card {
                background: #0b1826;
                border: 1px solid #1e3a5a;
                border-radius: 12px;
            }
            QLabel { background: transparent; color: #e8edf5; }
            QPushButton {
                border-radius: 7px; font-size: 13px; font-weight: bold;
                padding: 10px 20px; border: none;
            }
            QPushButton#play_btn { background: #1565c0; color: white; }
            QPushButton#play_btn:hover { background: #1e7ae0; }
            QPushButton#fav_btn { background: #111f30; color: #8aaac8; border: 1px solid #1e3a5a; }
            QPushButton#fav_btn:hover { background: #192a3e; }
            QPushButton#fav_btn[fav="1"] { background: #6b4c00; color: #ffd54f; border: 1px solid #c8860b; }
            QPushButton#fav_btn[fav="1"]:hover { background: #8a6200; }
        """)

        outer = QVBoxLayout(self)
        outer.setContentsMargins(0, 0, 0, 0)
        outer.addStretch()

        plot = (synopsis.strip() or
                (show_plot and (s.get("plot") or s.get("synopsis") or
                                s.get("description") or s.get("overview") or "").strip()) or "")
```

```
card = QWidget(objectName="card")
card.setFixedWidth(540 if plot else 380)
outer.addWidget(card, 0, Qt.AlignmentFlag.AlignHCenter)
outer.addStretch()

vl = QVBoxLayout(card)
vl.setContentsMargins(18, 16, 18, 16)
vl.setSpacing(0)

# — Title —————
name = s.get("name", "")
title_lbl = QLabel(name)
title_lbl.setWordWrap(True)
f = QFont(); f.setPixelSize(16); f.setBold(True)
title_lbl.setFont(f)
title_lbl.setStyleSheet("color: #ffffff;")
vl.addWidget(title_lbl)
vl.addSpacing(6)

# — Badge pills —————
badges = []
year = (s.get("releaseDate") or s.get("year") or "")
if year:
    badges.append(str(year)[:4])
if quality and quality not in ("N/A",):
    badges.append(quality)
genre = (s.get("genre") or "").split(",")[0].strip()
if genre and len(genre) < 25:
    badges.append(genre)

if badges:
    badge_row = QHBoxLayout()
    badge_row.setSpacing(6)
    badge_row.setContentsMargins(0, 0, 0, 0)
    badge_row.setAlignment(Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignVCenter)
    for txt in badges:
        lbl = QLabel(txt)
        lbl.setStyleSheet(
            "background:#14243a; color:#7ab0d8; font-size:11px; font-weight:bold;"
            " padding:2px 8px; border-radius:4px; border:1px solid #1e3a5a;"
        )
        badge_row.addWidget(lbl)
    badge_row.addStretch()
    vl.addLayout(badge_row)
    vl.addSpacing(10)

# — Divider —————
div = QFrame()
div.setFrameShape(QFrame.Shape.HLine)
div.setFixedHeight(1)
div.setStyleSheet("background:#1a3050;")
vl.addWidget(div)
vl.addSpacing(12)

# — Poster + synopsis —————
top = QHBoxLayout()
top.setSpacing(14)
top.setContentsMargins(0, 0, 0, 0)
```

```
base = item.data(BASE_POSTER_ROLE) if item else None
if (not base or base.isNull()) and item:
    ico = item.icon()
    if not ico.isNull():
        base = ico.pixmap(120, 120)

if base and not base.isNull():
    is_square = abs(base.width() - base.height()) < max(base.width(), base.height()) * 0.15
    tw, th = (110, 110) if is_square else (110, 160)
    thumb_lbl = QLabel()
    thumb_lbl.setFixedSize(tw, th)
    thumb_lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
    thumb_lbl.setPixmap(base.scaled(tw, th,
        Qt.AspectRatioMode.KeepAspectRatio,
        Qt.TransformationMode.SmoothTransformation))
    thumb_lbl.setStyleSheet("background:#060e18; border-radius:6px;")
    if plot:
        top.addWidget(thumb_lbl, 0, Qt.AlignmentFlag.AlignTop)
    else:
        top.addStretch()
        top.addWidget(thumb_lbl, 0, Qt.AlignmentFlag.AlignVCenter)
        top.addStretch()

if plot:
    max_chars = 340
    display_plot = plot if len(plot) <= max_chars else plot[:max_chars - 3] + "..."
    syn_lbl = QLabel(display_plot)
    syn_lbl.setWordWrap(True)
    syn_lbl.setAlignment(Qt.AlignmentFlag.AlignTop | Qt.AlignmentFlag.AlignLeft)
    syn_lbl.setStyleSheet("color:#9fb3c8; font-size:12px; background:transparent;")
    top.addWidget(syn_lbl, 1, Qt.AlignmentFlag.AlignTop)

vl.addLayout(top)
vl.addSpacing(14)

# — Divider —————
div2 = QFrame()
div2.setFrameShape(QFrame.Shape.HLine)
div2.setFixedHeight(1)
div2.setStyleSheet("background:#1a3050;")
vl.addWidget(div2)
vl.addSpacing(10)

# — Buttons —————
btn_row = QHBoxLayout()
btn_row.setSpacing(8)

self._fav_btn = QPushButton()
self._fav_btn.setObjectName("fav_btn")
self._fav_btn.setFocusPolicy(Qt.FocusPolicy.NoFocus)
self._update_fav_btn()
self._fav_btn.clicked.connect(self._on_fav)
btn_row.addWidget(self._fav_btn, 1)

play_btn = QPushButton("► Play")
play_btn.setObjectName("play_btn")
play_btn.setDefault(True)
play_btn.setAutoDefault(True)
play_btn.clicked.connect(self._on_play)
btn_row.addWidget(play_btn, 1)
```

```
        vl.addLayout(btn_row)
        play_btn.setFocus()

    def paintEvent(self, _):
        p = QPainter(self)
        p.fillRect(self.rect(), QColor(0, 0, 0, 160))
        p.end()

    def mousePressEvent(self, event):
        card = self.findChild(QWidget, "card")
        if card and not card.geometry().contains(event.pos()):
            self.reject()
        else:
            super().mousePressEvent(event)

    def _update_fav_btn(self):
        if self._is_fav:
            self._fav_btn.setText("★ Remove from Favorites")
            self._fav_btn.setProperty("fav", "1")
        else:
            self._fav_btn.setText("☆ Add to Favorites")
            self._fav_btn.setProperty("fav", "0")
        self._fav_btn.style().unpolish(self._fav_btn)
        self._fav_btn.style().polish(self._fav_btn)

    def _on_fav(self):
        self._is_fav = not self._is_fav
        self.fav_toggled = True
        self.accept()

    def _on_play(self):
        self.play_clicked = True
        self.accept()

    def keyPressEvent(self, event):
        if event.key() == Qt.Key.Key_Escape:
            self.reject()
        else:
            super().keyPressEvent(event)

class MainWindow(QMainWindow):
    def __init__(self, config: Config):
        super().__init__()
        self.config = config
        self.api: XtreamAPI | None = None
        self.pool = QThreadPool.globalInstance()
        self._tab = 0
        self._playing_stream: dict | None = None
        self._load_gen = 0
        self._auto_favorites = False
        self._known_live_cat_ids: set = set() # populated when Live TV categories load
        self._filter_uncategorized: bool = False
        self._filter_hidden: bool = False
        self._filter_working: bool = False
        self._filter_recently_added: bool = False
        self._all_cats: list = []
        self._cat_select_mode: bool = False
        self._ch_select_mode: bool = False
```

```
self._img_cache: dict[str, QPixmap] = {}
self._ch_delegate = ChannelDelegate()

# sleep timer
self._sleep_timer = QTimer(self)
self._sleep_timer.setInterval(1000)
self._sleep_timer.timeout.connect(self._sleep_tick)
self._sleep_remaining = 0

# progress tracking (continue watching)
self._progress_timer = QTimer(self)
self._progress_timer.setInterval(30_000)
self._progress_timer.timeout.connect(self._save_progress)
self._buf_timer = QTimer(self)
self._buf_timer.setInterval(1_000)
self._buf_timer.timeout.connect(self._poll_buffer)
self._buf_stall_ticks = 0
self._buf_last_pos = 0.0
self._buf_started = False
self._buf_stream_ticks = 0
self._failure_recorded = False
self._success_cleared = False
self._current_vod_key: str | None = None

# mini player
self._mini_player: _MiniPlayer | None = None

# series next-episode tracking
self._series_state: dict | None = None

# external player process (VLC, MPV standalone, etc.)
self._ext_proc = None
self._vlc_embedded = False
self._vlc_paused = False
self._vlc_monitor = None
self._hub_fs = False
self._pre_hub_fs_state = None

# VLC click detection (Windows only): poll GetAsyncKeyState every 50ms
# because VLC's child HWND swallows Win32 mouse events before Qt sees them.
self._vlc_btn_was_down = False
self._vlc_last_click = 0.0
self._vlc_click_timer = QTimer(self)
self._vlc_click_timer.setInterval(50)
self._vlc_click_timer.timeout.connect(self._poll_vlc_click)

# seek bar
self._seek_dragging = False
self._pos_timer = QTimer(self)
self._pos_timer.setInterval(500)
self._pos_timer.timeout.connect(self._update_seek_bar)

self.setWindowTitle("TVS Hub")
self.setMinimumSize(1100, 660)
self.resize(1340, 760)
self.setStyleSheet(_STYLE)

self._img_queue: list = []
self._img_active = 0
self._epg_queue: list = []
```

```

self._epg_active = 0
self._season_queue: list = []
self._season_active: int = 0
self._season_cache: dict = self._load_season_cache()
self._vod_q_queue: list = []
self._vod_q_active: int = 0
self._vod_q_cache: dict = self._load_vod_quality_cache()
self._live_q_queue: list = []
self._live_q_active: int = 0
self._live_q_cache: dict = self._load_quality_cache()
self._radio_streams_cache: list = []
self._radio_all_streams: list = [] # cached radio list (pre-hidden-filter) for ALL view

self._log_panel = LogPanel(self)
log = AppLogger.get()
log.entry_added.connect(self._log_panel.append)
log.info("TVS Hub started")

# spinner state
self._spin_idx = 0
self._spin_widgets: list[QLabel] = []
self._spin_timer = QTimer(self)
self._spin_timer.setInterval(80)
self._spin_timer.timeout.connect(self._tick_spinners)

self._build_ui()

# create spinner overlays now that list widgets exist
self._cat_spinner = self._make_spinner(self.cat_list)
self._ch_spinner = self._make_spinner(self.ch_list)
self._grid_spinner = self._make_spinner(self.poster_grid)

self._setup_shortcuts()
QTimer.singleShot(0, self._init_connection)

def _is_radio(self, s: dict) -> bool:
    """Return True if stream is radio – by metadata or learned from playback."""
    if _is_radio_stream(s):
        return True
    return str(s.get("stream_id", "")) in self.config.radio_stream_ids

@property
def _video_playing(self) -> bool:
    """True when a non-radio video stream is playing (live TV, movie, or series)."""
    return (self._playing_stream is not None
            and not self._is_radio(self._playing_stream))

def _dn(self, name: str) -> str:
    """Display name – strips country prefix when the user preference is on."""
    return _clean_name(name) if getattr(self.config, "hide_channel_prefixes", True) else name

# — build —————

def _build_ui(self):
    self._stack = QStackedWidget()
    self.setCentralWidget(self._stack)

    # Page 0 – home
    self._home = HomeScreen()
    self._home.section_clicked.connect(self._go_to_section)

```



```
self._home.login_clicked.connect(self._show_login)
self._home.logoff_clicked.connect(self._logoff)
self._home.epg_clicked.connect(self._open_epg)
self._home.multiscreen_clicked.connect(self._open_multiscreen)
self._home.catchup_clicked.connect(self._open_catchup)
self._home.radio_clicked.connect(self._open_radio)
self._home.settings_clicked.connect(self._open_player_settings)
self._home.backup_clicked.connect(self._open_backup_dialog)
self._stack.addWidget(self._home)

# Page 1 – content browser
self._stack.addWidget(self._build_content_page())
self.player.clicked.connect(self._on_player_clicked)
self.player.double_clicked.connect(self._on_player_double_clicked)
self.player.audio_only_changed.connect(self._on_audio_only)
self.player.resolution_changed.connect(self._on_resolution)

# Page 2 – EPG guide
self._epg_screen = EPGSscreen()
self._epg_screen.set_config(self.config)
self._epg_screen.back_to_home.connect(lambda: self._stack.setCurrentIndex(0))
self._epg_screen.channel_activated.connect(
    lambda s, url: AppLogger.get().info(f"EPG playing: {s.get('name','')}")
)
self._stack.addWidget(self._epg_screen)

# Page 3 – Multi-screen
self._multi_screen = MultiScreen()
self._multi_screen.set_config(self.config)
self._multi_screen.back_to_home.connect(lambda: self._stack.setCurrentIndex(0))
self._stack.addWidget(self._multi_screen)

# Page 4 – Player settings
from .player_select_screen import PlayerSelectScreen
self._player_settings = PlayerSelectScreen(self.config)
self._player_settings.back_clicked.connect(lambda: self._stack.setCurrentIndex(0))
self._stack.addWidget(self._player_settings)

# Page 5 – Radio station picker (category-style grid → EPG grid on click)
self._radio_screen = RadioScreen()
self._radio_screen.set_config(self.config)
self._radio_screen.back_to_home.connect(lambda: self._stack.setCurrentIndex(0))
self._radio_screen.channel_activated.connect(lambda s, url: self._play(s, url))
self._radio_screen.stop_requested.connect(self._stop)
self._stack.addWidget(self._radio_screen)

def _open_player_settings(self):
    self._player_settings.open()
    self._stack.setCurrentIndex(4)

def _open_backup_dialog(self):
    import json as _json
    from dataclasses import asdict
    from PyQt6.QtWidgets import QFileDialog, QMessageBox
    from .config import Account

    _BACKUP_FIELDS = [
        "favorite_streams", "cat_order",
        "hidden_stream_ids", "hidden_category_ids",
        "radio_favorite_ids",
```

```
]

dlg = QDialog(self)
dlg.setWindowTitle("Backup & Restore")
dlg.setMinimumWidth(340)
dlg.setStyleSheet(
    f"QDialog{{background:#0d1117;color:#d0d8e8;}}"
    f"QLabel{{color:#d0d8e8;background:transparent;}}"
)
vl = QVBoxLayout(dlg)
vl.setSpacing(12)
vl.setContentsMargins(20, 20, 20, 20)

title = QLabel("Backup & Restore Customizations")
tf = QFont(); tf.setPixelSize(14); tf.setBold(True)
title.setFont(tf)
title.setStyleSheet("color:white;")
vl.addWidget(title)

desc = QLabel(
    "Saves or restores:\n"
    "• Favorites • Category order\n"
    "• Hidden channels & categories\n"
    "• Radio favorites"
)
desc.setStyleSheet("color:#4a6a8a; font-size:12px;")
vl.addWidget(desc)

_btn_style = (
    "QPushButton{background:#1976d2;color:white;border:none;border-radius:5px;"
    "font-size:13px;font-weight:bold;padding:10px 0;}"
    "QPushButton:hover{background:#1e88e5;}"
)
_btn_style2 = (
    "QPushButton{background:#2e7d32;color:white;border:none;border-radius:5px;"
    "font-size:13px;font-weight:bold;padding:10px 0;}"
    "QPushButton:hover{background:#388e3c;}"
)
_btn_style3 = (
    "QPushButton{background:#b71c1c;color:white;border:none;border-radius:5px;"
    "font-size:13px;font-weight:bold;padding:10px 0;}"
    "QPushButton:hover{background:#c62828;}"
)

exp_btn = QPushButton(" Export")
exp_btn.setStyleSheet(_btn_style)
exp_btn.setFixedHeight(42)
vl.addWidget(exp_btn)

imp_btn = QPushButton(" Import")
imp_btn.setStyleSheet(_btn_style2)
imp_btn.setFixedHeight(42)
vl.addWidget(imp_btn)

from .config import CONFIG_FILE
clr_btn = QPushButton(" Clear Config")
clr_btn.setStyleSheet(_btn_style3)
clr_btn.setFixedHeight(42)
clr_btn.setVisible(CONFIG_FILE.exists())
vl.addWidget(clr_btn)
```

```
status = QLabel("")
status.setStyleSheet("color:#42a5f5; font-size:11px;")
status.setWordWrap(True)
vl.addWidget(status)

def do_export():
    path, _ = QFileDialog.getSaveFileName(
        dlg, "Export Customizations", "tvshub_backup.json",
        "JSON Files (*.json)")
    if not path:
        return
    data = {"tvshub_backup": True, "version": 2}
    for f in _BACKUP_FIELDS:
        data[f] = getattr(self.config, f, None)
    try:
        with open(path, "w", encoding="utf-8") as fh:
            _json.dump(data, fh, indent=2)
        status.setText(f"✓ Exported to {path}")
    except Exception as e:
        status.setText(f"Export failed: {e}")

def do_import():
    path, _ = QFileDialog.getOpenFileName(
        dlg, "Import Customizations", "",
        "JSON Files (*.json)")
    if not path:
        return
    try:
        with open(path, encoding="utf-8") as fh:
            data = _json.load(fh)
        if not data.get("tvshub_backup"):
            status.setText("Not a valid TVS Hub backup file.")
            return
        for f in _BACKUP_FIELDS:
            if f in data:
                setattr(self.config, f, data[f])
        self.config.save()
        status.setText("✓ Imported successfully. Changes take effect next time you open a section.")
    except Exception as e:
        status.setText(f"Import failed: {e}")

def do_clear():
    confirm = QMessageBox.question(
        dlg, "Clear Config",
        "This will delete all saved settings, accounts, favorites, and history.\n\nAre you sure?",
        QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.Cancel,
    )
    if confirm != QMessageBox.StandardButton.Yes:
        return
    from .config import CONFIG_FILE, CONNECTION_FILE
    try:
        if CONFIG_FILE.exists():
            CONFIG_FILE.unlink()
        if CONNECTION_FILE.exists():
            CONNECTION_FILE.unlink()
        self.config.__init__()
        self.api = None
        self.account_btn.setText("Login")
        self._home.set_logged_out()
```

```
        clr_btn.hide()
        status.setText("✓ Config cleared. Restart or log in again.")
    except Exception as e:
        status.setText(f"Clear failed: {e}")

exp_btn.clicked.connect(do_export)
imp_btn.clicked.connect(do_import)
clr_btn.clicked.connect(do_clear)

close_btn = QPushButton("Close")
close_btn.setStyleSheet(
    "QPushButton{background:transparent;color:#4a6a8a;border:1px solid #1e2a3a;"
    "border-radius:5px;padding:6px 0;font-size:12px;}"
    "QPushButton:hover{color:white;border-color:#4a6a8a;}"
)
close_btn.clicked.connect(dlg.accept)
vl.addWidget(close_btn)

dlg.exec()

def _build_content_page(self) -> QWidget:
    page = QWidget()
    vl = QVBoxLayout(page)
    vl.setContentsMargins(0, 0, 0, 0)
    vl.setSpacing(0)

    self._content_topbar = self._build_topbar()
    vl.addWidget(self._content_topbar)

    # — back bar (shown when watching VOD) —————
    self._back_bar = QWidget()
    self._back_bar.setObjectName("back_bar")
    self._back_bar.setFixedHeight(36)
    bb = QHBoxLayout(self._back_bar)
    bb.setContentsMargins(12, 0, 12, 0)
    back_btn = QPushButton("← Back to browse")
    back_btn.setObjectName("back_btn")
    back_btn.clicked.connect(self._back_to_browse)
    bb.addWidget(back_btn)
    self._vod_title_lbl = QLabel("")
    self._vod_title_lbl.setStyleSheet("color:#fff; font-size:13px; font-weight:bold; background:transparent;")
    bb.addWidget(self._vod_title_lbl)
    bb.addStretch()
    self._back_bar.hide()
    vl.addWidget(self._back_bar)

    # — main splitter: [cat | middle | player] —————
    self._splitter = QSplitter(Qt.Orientation.Horizontal)
    self._splitter.setHandleWidth(1)
    self._splitter.setStyleSheet(f"QSplitter {{ border-bottom: 1px solid {_BORDER}; }}")

    self._splitter.addWidget(self._build_cat_panel())      # index 0
    self._splitter.addWidget(self._build_middle_panel())  # index 1
    self._splitter.addWidget(self._build_player_panel())  # index 2
    self._splitter.setSizes([210, 240, 800])

    vl.addWidget(self._splitter, stretch=1)

    # — seek bar —————
    vl.addWidget(self._build_seek_bar())
```

```
# — controls —————
self._controls_bar = self._build_controls()
vl.addWidget(self._controls_bar)

return page

def _build_topbar(self) -> QWidget:
    bar = QWidget()
    bar.setObjectName("topbar")
    bar.setFixedHeight(50)
    hl = QHBoxLayout(bar)
    hl.setContentsMargins(16, 0, 16, 0)
    hl.setSpacing(8)

    home_btn = QPushButton("△")
    home_btn.setObjectName("icon_btn")
    home_btn.setFixedSize(32, 32)
    home_btn.setToolTip("Home")
    home_btn.clicked.connect(self._go_home)
    hl.addWidget(home_btn)

    div = QLabel("|")
    div.setStyleSheet(f"color:{_BORDER}; font-size:22px; background:transparent;")
    hl.addWidget(div)

    self._section_lbl = QLabel("Live TV")
    sf = QFont(); sf.setPixelSize(15); sf.setBold(True)
    self._section_lbl.setFont(sf)
    self._section_lbl.setStyleSheet("color:white; background:transparent;")
    hl.addWidget(self._section_lbl)

    hl.addSpacing(16)

    # Tab buttons
    self._tab_btns: list[QPushButton] = []
    for i, label in enumerate(["Live TV", "Movies", "Series", "Radio"]):
        btn = QPushButton(label)
        btn.setObjectName("tab_btn")
        btn.setCheckable(True)
        btn.setChecked(i == 0)
        btn.clicked.connect(lambda _, idx=i: self._on_tab_changed(idx))
        self._tab_btns.append(btn)
        hl.addWidget(btn)

    hl.addStretch()

    self._topbar_time = QLabel("")
    self._topbar_time.setStyleSheet(f"color:{_DIM}; font-size:12px; background:transparent;")
    hl.addWidget(self._topbar_time)
    self._tick_timer = QTimer(bar)
    self._tick_timer.timeout.connect(self._update_topbar_time)
    self._tick_timer.start(1000)
    self._update_topbar_time()

    hl.addSpacing(12)

    self._epg_refresh_btn = QPushButton("")
    self._epg_refresh_btn.setObjectName("icon_btn")
    self._epg_refresh_btn.setFixedSize(28, 28)
```

```
self._epg_refresh_btn.setToolTip("Refresh EPG data")
self._epg_refresh_btn.clicked.connect(self._refresh_epg)
self._epg_refresh_btn.hide() # shown only on Live TV tab
hl.addWidget(self._epg_refresh_btn)

topbar_gear_btn = QPushButton("⚙")
topbar_gear_btn.setObjectName("icon_btn")
topbar_gear_btn.setFixedSize(28, 28)
topbar_gear_btn.setToolTip("Player Settings")
topbar_gear_btn.clicked.connect(self._open_player_settings)
hl.addWidget(topbar_gear_btn)

search_btn = QPushButton("")
search_btn.setObjectName("icon_btn")
search_btn.setFixedSize(28, 28)
search_btn.setToolTip("Search all channels [Ctrl+F]")
search_btn.clicked.connect(self._open_global_search)
hl.addWidget(search_btn)

self.account_btn = QPushButton("Login")
self.account_btn.setObjectName("login_btn")
self.account_btn.clicked.connect(self._show_login)
hl.addWidget(self.account_btn)

return bar

def _build_cat_panel(self) -> QWidget:
    panel = QWidget()
    panel.setObjectName("cat_panel")
    vl = QVBoxLayout(panel)
    vl.setContentsMargins(0, 0, 0, 0)
    vl.setSpacing(0)

    sw = QWidget()
    sw.setFixedHeight(44)
    sl = QHBoxLayout(sw)
    sl.setContentsMargins(8, 5, 6, 5)
    sl.setSpacing(5)
    self.search_edit = QLineEdit()
    self.search_edit.setPlaceholderText("Search in categories...")
    self.search_edit.textChanged.connect(self._filter_content)
    sl.addWidget(self.search_edit, stretch=1)

    self._cat_select_btn = QPushButton("🔍")
    self._cat_select_btn.setFixedSize(28, 28)
    self._cat_select_btn.setCheckable(True)
    self._cat_select_btn.setToolTip("Select categories to hide")
    self._cat_select_btn.setStyleSheet(
        "QPushButton{background:transparent;color:#4a6a8a;border:1px solid #1e2a3a;"
        "border-radius:4px;font-size:14px;}"
        "QPushButton:hover{color:#fff;border-color:#4a6a8a;}"
        "QPushButton:checked{background:#1976d2;color:#fff;border-color:#1976d2;}"
    )
    self._cat_select_btn.clicked.connect(self._toggle_cat_select_mode)
    sl.addWidget(self._cat_select_btn)
    vl.addWidget(sw)

    self.cat_list = _CatList()
    self.cat_list.setObjectName("cat_list")
    self.cat_list.setDragDropMode(QListWidget.DragDropMode.InternalMove)
```

```

self.cat_list.setDefaultDropAction(Qt.DropAction.MoveAction)
self.cat_list.setDragDropOverwriteMode(False)
self.cat_list.itemClicked.connect(self._on_category_clicked)
self.cat_list.model().rowsMoved.connect(lambda *_: self._on_cat_reordered())
self.cat_list.setContextMenuPolicy(Qt.ContextMenuPolicy.CustomContextMenu)
self.cat_list.customContextMenuRequested.connect(self._cat_context_menu)
vl.addWidget(self.cat_list, stretch=1)

# action bar – shown only in select mode
self._cat_action_bar = QWidget()
self._cat_action_bar.setFixedHeight(40)
self._cat_action_bar.setStyleSheet(
    "background:#0d1117; border-top:1px solid #1e2a3a;"
)
al = QHBoxLayout(self._cat_action_bar)
al.setContentsMargins(8, 4, 8, 4)
al.setSpacing(6)
self._cat_hide_btn = QPushButton("Hide 0 Selected")
self._cat_hide_btn.setEnabled(False)
self._cat_hide_btn.setStyleSheet(
    "QPushButton{background:#c62828;color:white;border:none;border-radius:4px;"
    "font-size:11px;font-weight:bold;padding:4px 8px;}"
    "QPushButton:hover{background:#e53935;}"
    "QPushButton:disabled{background:#1e2a3a;color:#4a6a8a;}"
)
self._cat_hide_btn.clicked.connect(self._hide_selected_categories)
al.addWidget(self._cat_hide_btn, stretch=1)
_cancel_btn = QPushButton("x")
_cancel_btn.setFixedSize(28, 28)
_cancel_btn.setToolTip("Cancel selection")
_cancel_btn.setStyleSheet(
    "QPushButton{background:transparent;color:#4a6a8a;border:none;font-size:14px;}"
    "QPushButton:hover{color:#fff;}"
)
_cancel_btn.clicked.connect(self._exit_cat_select_mode)
al.addWidget(_cancel_btn)
self._cat_action_bar.hide()
vl.addWidget(self._cat_action_bar)

return panel

def _build_middle_panel(self) -> QWidget:
    self._middle_stack = QStackedWidget()

    # index 0 – channel list (Live TV)
    ch_wrap = QWidget()
    ch_wrap.setObjectName("ch_panel")
    cvl = QVBoxLayout(ch_wrap)
    cvl.setContentsMargins(0, 0, 0, 0)
    cvl.setSpacing(0)
    ch_header_bar = QWidget()
    ch_header_bar.setFixedHeight(28)
    ch_header_bar.setStyleSheet(f"background:{_BG};")
    ch_hl = QHBoxLayout(ch_header_bar)
    ch_hl.setContentsMargins(14, 0, 4, 0)
    ch_hl.setSpacing(4)
    self._ch_header = QLabel("CHANNELS")
    self._ch_header.setStyleSheet(
        f"color:{_DIM}; font-size:10px; font-weight:bold; background:transparent;"
    )
    ch_hl.addWidget(self._ch_header, stretch=1)
    self._ch_select_btn = QPushButton("☐")

```

```

self._ch_select_btn.setFixedSize(22, 22)
self._ch_select_btn.setCheckable(True)
self._ch_select_btn.setToolTip("Select channels to hide")
self._ch_select_btn.setStyleSheet(
    "QPushButton{background:transparent;color:#4a6a8a;border:1px solid #1e2a3a;"
    "border-radius:3px;font-size:12px;}"
    "QPushButton:hover{color:#fff;border-color:#4a6a8a;}"
    "QPushButton:checked{background:#1976d2;color:#fff;border-color:#1976d2;}"
)
self._ch_select_btn.clicked.connect(self._toggle_ch_select_mode)
ch_hl.addWidget(self._ch_select_btn)
cvl.addWidget(ch_header_bar)
self.ch_list = QListWidget()
self.ch_list.setItemDelegate(self._ch_delegate)
self.ch_list.setIconSize(QSize(_LOGO_SZ, _LOGO_SZ))
self.ch_list.itemClicked.connect(self._on_channel_activated)
self.ch_list.setContextMenuPolicy(Qt.ContextMenuPolicy.CustomContextMenu)
self.ch_list.customContextMenuRequested.connect(self._ch_context_menu)
self.ch_list.viewport().installEventFilter(self)
cvl.addWidget(self.ch_list, stretch=1)

# channel action bar – shown only in select mode
self._ch_action_bar = QWidget()
self._ch_action_bar.setFixedHeight(40)
self._ch_action_bar.setStyleSheet(
    "background:#0d1117; border-top:1px solid #1e2a3a;"
)
ch_al = QHBoxLayout(self._ch_action_bar)
ch_al.setContentsMargins(8, 4, 8, 4)
ch_al.setSpacing(6)
self._ch_hide_btn = QPushButton("Hide 0 Selected")
self._ch_hide_btn.setEnabled(False)
self._ch_hide_btn.setStyleSheet(
    "QPushButton{background:#c62828;color:white;border:none;border-radius:4px;"
    "font-size:11px;font-weight:bold;padding:4px 8px;}"
    "QPushButton:hover{background:#e53935;}"
    "QPushButton:disabled{background:#1e2a3a;color:#4a6a8a;}"
)
self._ch_hide_btn.clicked.connect(self._hide_selected_channels)
ch_al.addWidget(self._ch_hide_btn, stretch=1)
_ch_cancel_btn = QPushButton("x")
_ch_cancel_btn.setFixedSize(28, 28)
_ch_cancel_btn.setToolTip("Cancel selection")
_ch_cancel_btn.setStyleSheet(
    "QPushButton{background:transparent;color:#4a6a8a;border:none;font-size:14px;}"
    "QPushButton:hover{color:#fff;}"
)
_ch_cancel_btn.clicked.connect(self._exit_ch_select_mode)
ch_al.addWidget(_ch_cancel_btn)
self._ch_action_bar.hide()
cvl.addWidget(self._ch_action_bar)
self._middle_stack.addWidget(ch_wrap) # index 0

# index 1 – poster grid (Movies / Series)
self.poster_grid = QListWidget()
self.poster_grid.setObjectName("grid_list")
self.poster_grid.setViewMode(QListWidget.ViewMode.IconMode)
self.poster_grid.setIconSize(QSize(_POSTER_W, _POSTER_H))
self.poster_grid.setGridSize(QSize(_POSTER_W + 14, _POSTER_H + 6))
self.poster_grid.setResizeMode(QListWidget.ResizeMode.Adjust)
self.poster_grid.setSpacing(5)

```



```
self.poster_grid.setWordWrap(False)
self._poster_delegate = PosterDelegate(_POSTER_W, _POSTER_H, self.poster_grid)
self.poster_grid.setItemDelegate(self._poster_delegate)
self.poster_grid.itemClicked.connect(self._on_vod_activated)
self.poster_grid.setContextMenuPolicy(Qt.ContextMenuPolicy.CustomContextMenu)
self.poster_grid.customContextMenuRequested.connect(self._poster_context_menu)
self.poster_grid.viewport().installEventFilter(self)

# Wrap poster_grid + probe status bar in a container
_grid_wrap = QWidget()
_grid_wrap.setStyleSheet("background: transparent;")
_gvl = QVBoxLayout(_grid_wrap)
_gvl.setContentsMargins(0, 0, 0, 0)
_gvl.setSpacing(0)
_gvl.addWidget(self.poster_grid)
self._probe_status_lbl = QLabel("")
self._probe_status_lbl.setStyleSheet(
    "background:#07111c; color:#4a7a9a; font-size:11px; padding:3px 10px;"
    " border-top: 1px solid #0f2030;"
)
self._probe_status_lbl.setVisible(False)
_gvl.addWidget(self._probe_status_lbl)
self._middle_stack.addWidget(_grid_wrap) # index 1

return self._middle_stack

def _build_player_panel(self) -> QWidget:
    panel = QWidget()
    vl = QVBoxLayout(panel)
    vl.setContentsMargins(0, 0, 0, 0)
    vl.setSpacing(0)

    self._player_stack = QStackedWidget()

    # page 0 - idle placeholder
    idle_w = QWidget()
    idle_w.setStyleSheet("background:#050a10;")
    idle_vl = QVBoxLayout(idle_w)
    idle_vl.setContentsMargins(0, 0, 0, 0)
    idle_vl.addStretch()
    idle_icon = QLabel("")
    idle_icon.setAlignment(Qt.AlignmentFlag.AlignCenter)
    idle_icon.setStyleSheet("font-size:52px; background:transparent; color:#1a2a3a;")
    idle_vl.addWidget(idle_icon)
    idle_lbl = QLabel("Select a channel to start watching")
    idle_lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
    idle_lbl.setStyleSheet("color:#1e3050; font-size:14px; background:transparent; margin-top:10px;")
    idle_vl.addWidget(idle_lbl)
    idle_vl.addStretch()
    self._player_stack.addWidget(idle_w) # index 0

    # page 1 - MPV player
    self.player = PlayerWidget()
    self._player_stack.addWidget(self.player) # index 1
    self._player_stack.setCurrentIndex(0)

    vl.addWidget(self._player_stack, stretch=1)
    vl.addWidget(self._build_epg_bar())
    return panel
```

```
def _build_epg_bar(self) -> QWidget:
    bar = QWidget()
    bar.setObjectName("epg_bar")
    bar.hide()
    vl = QVBoxLayout(bar)
    vl.setContentsMargins(14, 6, 14, 6)
    vl.setSpacing(4)

    hl = QHBoxLayout()
    hl.setContentsMargins(0, 0, 0, 0)
    self.epg_now = QLabel("")
    self.epg_now.setStyleSheet("color:#fff; font-size:13px; background:transparent;")
    hl.addWidget(self.epg_now)
    hl.addStretch()
    self.epg_next = QLabel("")
    self.epg_next.setStyleSheet(f"color:{_DIM}; font-size:13px; background:transparent;")
    hl.addWidget(self.epg_next)
    vl.addLayout(hl)

    self.epg_desc = QLabel("")
    self.epg_desc.setWordWrap(True)
    self.epg_desc.setMaximumHeight(72)
    self.epg_desc.setStyleSheet(f"color:{_DIM}; font-size:12px; background:transparent;")
    self.epg_desc.hide()
    vl.addWidget(self.epg_desc)

    self._epg_bar = bar
    return bar

def _build_seek_bar(self) -> QWidget:
    bar = QWidget()
    bar.setObjectName("controls")
    bar.setFixedHeight(34)
    bar.hide()
    hl = QHBoxLayout(bar)
    hl.setContentsMargins(16, 0, 16, 0)
    hl.setSpacing(10)

    self._pos_lbl = QLabel("0:00")
    self._pos_lbl.setFixedWidth(46)
    self._pos_lbl.setStyleSheet(f"color:{_TEXT}; font-size:11px; background:transparent;")

    self._seek_slider = QSlider(Qt.Orientation.Horizontal)
    self._seek_slider.setObjectName("seek_bar")
    self._seek_slider.setRange(0, 1000)
    self._seek_slider.setCursor(Qt.CursorShape.PointingHandCursor)
    self._seek_slider.sliderPressed.connect(lambda: setattr(self, '_seek_dragging', True))
    self._seek_slider.sliderReleased.connect(self._on_seek_released)

    self._dur_lbl = QLabel("0:00")
    self._dur_lbl.setFixedWidth(46)
    self._dur_lbl.setAlignment(Qt.AlignmentFlag.AlignRight | Qt.AlignmentFlag.AlignVCenter)
    self._dur_lbl.setStyleSheet(f"color:{_DIM}; font-size:11px; background:transparent;")

    hl.addWidget(self._pos_lbl)
    hl.addWidget(self._seek_slider, stretch=1)
    hl.addWidget(self._dur_lbl)

    self._seek_bar = bar
    return bar
```

```
def _build_controls(self) -> QWidget:
    bar = QWidget()
    bar.setObjectName("controls")
    bar.setFixedHeight(52)
    cl = QHBoxLayout(bar)
    cl.setContentsMargins(16, 0, 16, 0)
    cl.setSpacing(10)
    self._player_badge = QLabel("")
    self._player_badge.setStyleSheet(
        "color:#42a5f5; font-size:10px; font-weight:bold;"
        " background:transparent; border:none; padding:2px 4px;"
    )
    self._player_badge.hide()
    cl.addWidget(self._player_badge)

    self.now_playing = QLabel("No channel selected")
    self.now_playing.setStyleSheet("color:#fff; font-size:13px; font-weight:bold; background:transparent;")
    cl.addWidget(self.now_playing, stretch=1)

    self._buf_indicator = QLabel("● STREAM")
    self._buf_indicator.setAlignment(Qt.AlignmentFlag.AlignCenter)
    self._buf_indicator.setMinimumWidth(68)
    self._buf_indicator.setFixedHeight(22)
    self._buf_indicator.setStyleSheet(
        "color:#00c853; font-size:10px; font-weight:bold;"
        " background:#071a0d; border:1px solid #1a4f2a;"
        " border-radius:4px; padding:0 6px;"
    )
    self._buf_indicator.hide()
    cl.addWidget(self._buf_indicator)

    self.play_btn = QPushButton("")
    self.play_btn.setObjectName("icon_btn")
    self.play_btn.setFixedSize(32, 32)
    self.play_btn.clicked.connect(self._toggle_pause)
    cl.addWidget(self.play_btn)
    self.stop_btn = QPushButton("")
    self.stop_btn.setObjectName("icon_btn")
    self.stop_btn.setFixedSize(32, 32)
    self.stop_btn.clicked.connect(self._stop)
    cl.addWidget(self.stop_btn)
    _vol_icon = QLabel("")
    _vol_icon.setStyleSheet("background:transparent;")
    cl.addWidget(_vol_icon)
    self.vol_slider = QSlider(Qt.Orientation.Horizontal)
    self.vol_slider.setRange(0, 130)
    self.vol_slider.setValue(self.config.volume)
    self.vol_slider.setFixedWidth(90)
    self.vol_slider.setStyleSheet("background:transparent;")
    self.vol_slider.valueChanged.connect(self._on_volume)
    cl.addWidget(self.vol_slider)
    tracks_btn = QPushButton("")
    tracks_btn.setObjectName("icon_btn")
    tracks_btn.setFixedSize(32, 32)
    tracks_btn.setToolTip("Audio / Subtitle tracks")
    tracks_btn.clicked.connect(self._show_tracks_menu)
    cl.addWidget(tracks_btn)

    self._cc_btn = QPushButton("CC")
```

```
self._cc_btn.setObjectName("icon_btn")
self._cc_btn.setFixedSize(32, 32)
self._cc_btn.setToolTip("Toggle closed captions")
self._cc_btn.clicked.connect(self._toggle_cc)
cl.addWidget(self._cc_btn)

self._next_ep_btn = QPushButton(" Next")
self._next_ep_btn.setObjectName("icon_btn")
self._next_ep_btn.setFixedHeight(32)
self._next_ep_btn.setToolTip("Next episode")
self._next_ep_btn.clicked.connect(self._go_next_episode)
self._next_ep_btn.hide()
cl.addWidget(self._next_ep_btn)

info_btn = QPushButton("")
info_btn.setObjectName("icon_btn")
info_btn.setFixedSize(32, 32)
info_btn.setToolTip("Stream info  [I]")
info_btn.clicked.connect(self._show_stream_info)
cl.addWidget(info_btn)

self._sleep_btn = QPushButton("")
self._sleep_btn.setObjectName("icon_btn")
self._sleep_btn.setFixedSize(32, 32)
self._sleep_btn.setToolTip("Sleep timer")
self._sleep_btn.clicked.connect(self._toggle_sleep_timer)
cl.addWidget(self._sleep_btn)

mini_btn = QPushButton("⌵")
mini_btn.setObjectName("icon_btn")
mini_btn.setFixedSize(32, 32)
mini_btn.setToolTip("Pop out mini player")
mini_btn.clicked.connect(self._open_mini_player)
cl.addWidget(mini_btn)

self.fs_btn = QPushButton("")
self.fs_btn.setObjectName("icon_btn")
self.fs_btn.setFixedSize(32, 32)
self.fs_btn.setToolTip("Fullscreen  [F]")
self.fs_btn.clicked.connect(self._toggle_fullscreen)
cl.addWidget(self.fs_btn)
return bar

def _setup_shortcuts(self):
    QShortcut(QKeySequence("Space"), self).activated.connect(self._toggle_pause)
    QShortcut(QKeySequence("F"), self).activated.connect(self._toggle_fullscreen)
    QShortcut(QKeySequence("Ctrl+L"), self).activated.connect(self._show_login)
    QShortcut(QKeySequence("Ctrl+F"), self).activated.connect(self._open_global_search)
    QShortcut(QKeySequence("I"), self).activated.connect(self._show_stream_info)
    QShortcut(QKeySequence("Escape"), self).activated.connect(self._handle_escape)
    QShortcut(QKeySequence("Up"), self).activated.connect(self._nav_up)
    QShortcut(QKeySequence("Down"), self).activated.connect(self._nav_down)
    QShortcut(QKeySequence("Left"), self).activated.connect(self._nav_left)
    QShortcut(QKeySequence("Right"), self).activated.connect(self._nav_right)
    QShortcut(QKeySequence("Return"), self).activated.connect(self._nav_enter)
    QShortcut(QKeySequence("Enter"), self).activated.connect(self._nav_enter)

def _focused_list(self):
    """Return whichever navigable list currently has focus, or None."""
    for w in (self.cat_list, self.ch_list, self.poster_grid):
```

```
        if w.hasFocus():
            return w
        return None

def _nav_up(self):
    w = self._focused_list()
    if w is None:
        # Default: move in the content list
        w = self.ch_list if self._tab in (0, 3) else self.poster_grid
        w.setFocus()
    row = w.currentRow()
    if row > 0:
        w.setCurrentRow(row - 1)
        w.scrollToItem(w.currentItem())

def _nav_down(self):
    w = self._focused_list()
    if w is None:
        w = self.ch_list if self._tab in (0, 3) else self.poster_grid
        w.setFocus()
    row = w.currentRow()
    if row < w.count() - 1:
        w.setCurrentRow(row + 1)
        w.scrollToItem(w.currentItem())

def _nav_left(self):
    """Move focus to the category panel."""
    self.cat_list.setFocus()
    if self.cat_list.currentRow() < 0 and self.cat_list.count() > 0:
        self.cat_list.setCurrentRow(0)

def _nav_right(self):
    """Move focus to the content panel (channel list or poster grid)."""
    if self._tab in (0, 3):
        self.ch_list.setFocus()
        if self.ch_list.currentRow() < 0 and self.ch_list.count() > 0:
            self.ch_list.setCurrentRow(0)
    else:
        self.poster_grid.setFocus()
        if self.poster_grid.currentRow() < 0 and self.poster_grid.count() > 0:
            self.poster_grid.setCurrentRow(0)

def _nav_enter(self):
    """Activate the currently focused item."""
    w = self._focused_list()
    if w is None:
        return
    item = w.currentItem()
    if item is None:
        return
    if w is self.cat_list:
        self._on_category_clicked(item)
    elif w is self.ch_list:
        self._on_channel_activated(item)
    elif w is self.poster_grid:
        self._on_vod_activated(item)

# — navigation —————

def _go_to_section(self, tab_index: int):
```

```
self._on_tab_changed(tab_index)
self._stack.setCurrentIndex(1)

def _on_tab_changed(self, index: int):
    if self._cat_select_mode:
        self._exit_cat_select_mode()
    if self._ch_select_mode:
        self._exit_ch_select_mode()
    # Keep radio audio alive when switching between Live TV and Radio tabs.
    # Stop for everything else (VOD/Series, or external player, or non-radio streams).
    radio_playing = (
        self._playing_stream is not None
        and self._is_radio(self._playing_stream)
        and self._ext_proc is None
    )
    if (self._playing_stream is not None or self._ext_proc is not None) and not (
        radio_playing and index == 3
    ):
        self._stop()
    self._tab = index
    if index == 0:
        self._auto_favorites = True
    for i, btn in enumerate(self._tab_btns):
        btn.setChecked(i == index)
    titles = ["Live TV", "Movies", "Series", "Radio"]
    self._section_lbl.setText(titles[index])
    self._back_bar.hide()
    self._epg_bar.hide()
    self._next_ep_btn.hide()
    self._series_state = None

    if index in (0, 3):
        # 3-column: cat | channels | player
        self._middle_stack.setCurrentIndex(0)
        self._middle_stack.show()
        self._splitter.widget(2).show()
        self._splitter.setSizes([210, 240, 800])
        self._epg_refresh_btn.show()
    else:
        # 2-column: cat | poster grid
        self._middle_stack.setCurrentIndex(1)
        self._middle_stack.show()
        self._splitter.widget(2).hide()
        self._splitter.setSizes([210, 1000, 0])
        self._epg_refresh_btn.hide()

    if self.api:
        self._load_categories()

def _back_to_browse(self):
    """Return from VOD player to the poster grid."""
    self._save_progress()
    self._back_bar.hide()
    self._epg_bar.hide()
    self._next_ep_btn.hide()
    self._series_state = None
    self._middle_stack.show()
    self._splitter.widget(2).hide()
    self._splitter.setSizes([210, 1000, 0])
    if self._ext_proc is not None:
```

```

        try:
            if self._ext_proc.poll() is None:
                self._ext_proc.terminate()
        except Exception:
            pass
        self._ext_proc = None
    if self._hub_fs:
        self._toggle_hub_fs()
    self._vlc_embedded = False
    self.player.stop()
    self._player_stack.setCurrentIndex(0)
    self._playing_stream = None

def _on_audio_only(self, is_audio_only: bool):
    if not self._playing_stream or self._ext_proc is not None:
        return
    self.player.hide_message() # stream has loaded
    if is_audio_only:
        sid = str(self._playing_stream.get("stream_id", ""))
        if sid and sid not in self.config.radio_stream_ids:
            self.config.radio_stream_ids.append(sid)
            self.config.save()
            AppLogger.get().info(
                f"Learned radio (audio-only): {self._playing_stream.get('name','')}"
            )

def _on_player_clicked(self):
    self._toggle_fullscreen()

def _on_player_double_clicked(self):
    pass

def _poll_vlc_click(self):
    """Detect clicks on embedded VLC on Windows via GetAsyncKeyState polling.
    VLC's child HWND swallows Win32 mouse events before Qt sees them, so
    mousePressEvent never fires over the video area."""
    if not self._vlc_embedded or sys.platform != "win32":
        return
    import ctypes, ctypes.wintypes as wt, time as _time
    state = ctypes.windll.user32.GetAsyncKeyState(0x01) # VK_LBUTTON
    pressed = bool(state & 0x8000)
    if pressed and not self._vlc_btn_was_down:
        # New press - check cursor is over the player widget.
        pt = wt.POINT()
        ctypes.windll.user32.GetCursorPos(ctypes.byref(pt))
        tl = self.player.mapToGlobal(self.player.rect().topLeft())
        from PyQt6.QtCore import QRect
        player_rect = QRect(tl.x(), tl.y(), self.player.width(), self.player.height())
        if player_rect.contains(pt.x, pt.y):
            now = _time.monotonic()
            if now - self._vlc_last_click > 0.4: # debounce
                self._vlc_last_click = now
                self._toggle_fullscreen()
    self._vlc_btn_was_down = pressed

def mousePressEvent(self, event):
    if self._hub_fs and event.button() == Qt.MouseButton.LeftButton:
        click_in_player = self.player.rect().contains(
            self.player.mapFromGlobal(event.globalPosition()).toPoint()
        )

```

```
        if not click_in_player:
            self._toggle_hub_fs()
        super().mousePressEvent(event)

def _handle_escape(self):
    if self._hub_fs:
        self._toggle_hub_fs()
    elif self._back_bar.isVisible():
        self._back_to_browse()

# — auth —————

def _init_connection(self):
    if self.config.has_account:
        from .login_dialog import _SERVER
        self._connect(_SERVER,
                      self.config.account.username,
                      self.config.account.password)
    else:
        self._show_login(force=True)

def _show_login(self, force: bool = False):
    dlg = LoginDialog(self.config, parent=self)
    if dlg.exec():
        s, u, p = dlg.credentials()
        if s and u:
            self._connect(s, u, p)
    elif force:
        QTimer.singleShot(0, self.close)

def _connect(self, server, username, password):
    self.api = XtreamAPI(server, username, password)
    self._status(f"Connecting to {server} ...")
    w = Worker(self.api.authenticate)
    w.signals.result.connect(self._on_auth_ok)
    w.signals.error.connect(self._on_auth_fail)
    self.pool.start(w)

def _on_auth_ok(self, data: dict):
    info = data.get("user_info", {})
    user = info.get("username", self.api.username)
    self.account_btn.setText((user[:14] + "...") if len(user) > 14 else user)
    self._status(f"Connected as {user}")
    self.config.account = Account(username=self.api.username,
                                   password=self.api.password)
    self.config.save()

    self._home.set_username(user)
    self._epg_screen.set_api(self.api)
    self._multi_screen.set_api(self.api)
    self._radio_screen.set_api(self.api)

    # save to accounts list for quick re-login
    saved = Account(username=self.api.username,
                    password=self.api.password, label=self.api.username)
    self.config.save_account(saved)

    exp_ts = info.get("exp_date")
    if exp_ts:
        try:
```



```

exp_dt    = datetime.fromtimestamp(int(exp_ts))
exp_str    = exp_dt.strftime("%B %d, %Y")
days_left = (exp_dt - datetime.now()).days
self._home.set_expiry(exp_str)
if days_left <= 7:
    self._home.warn_expiry(days_left)
    if days_left <= 0:
        msg = "Your subscription has expired. Please renew to continue watching."
    elif days_left == 1:
        msg = "Your subscription expires tomorrow!\n\nPlease renew to avoid interruption."
    else:
        msg = (f"Your subscription expires on {exp_str} "
                f"({days_left} days left).\n\nPlease renew to avoid interruption.")
    def _show_renew_dialog(m=msg):
        box = QMessageBox(self)
        box.setWindowTitle("Subscription Expiring Soon")
        box.setIcon(QMessageBox.Icon.Warning)
        box.setText(m)
        renew = box.addButton("Renew Now", QMessageBox.ButtonRole.AcceptRole)
        box.addButton("Dismiss", QMessageBox.ButtonRole.RejectRole)
        box.exec()
        if box.clickedButton() == renew:
            import webbrowser
            webbrowser.open("https://tvstreams.ca/order")
    QTimer.singleShot(600, _show_renew_dialog)
except Exception:
    pass

self._stack.setCurrentIndex(0)
# Pre-cache quality badges for the Canadian category in the background
QTimer.singleShot(500, self._precache_canada_quality)
# Pre-cache quality badges for the 30 most-recent movies and series
QTimer.singleShot(3000, self._precache_vod_quality)

def _precache_canada_quality(self):
    """Fetch Canadian channels and probe quality in the background at startup."""
    if not self.api:
        return

    def _fetch():
        cats = self.api.get_live_categories()
        canada = next(
            (c for c in cats
             if "CANAD" in (c.get("category_name") or "").upper()),
            None
        )
        if not canada:
            return []
        return self.api.get_live_streams(canada["category_id"])

    w = Worker(_fetch)
    w.signals.result.connect(self._on_precache_streams)
    w.signals.error.connect(lambda _: None)
    self.pool.start(w)

def _on_precache_streams(self, streams: list):
    """Queue quality probes for uncached Canadian streams silently."""
    uncached = [
        s for s in streams
        if s.get("stream_id")

```

```

        and str(s.get("stream_id", "")) not in self._live_q_cache
        and not self._is_radio(s)
    ]
    for s in uncached:
        sid = str(s.get("stream_id", ""))
        name = s.get("name", "")
        url = self.api.live_url(sid)
        hint = _stream_quality_hint(s)

    def _probe(url=url, hint=hint, sid=sid):
        import subprocess, json as _j, shutil, re as _re
        if shutil.which("ffprobe"):
            try:
                res = subprocess.run(
                    ["ffprobe", "-v", "quiet", "-print_format", "json",
                     "-show_streams", "-select_streams", "v:0",
                     "-timeout", "8000000", url],
                    capture_output=True, text=True, timeout=12,
                )
                data = _j.loads(res.stdout)
                st = data.get("streams", [])
                if st:
                    ww = int(st[0].get("width", 0) or 0)
                    hh = int(st[0].get("height", 0) or 0)
                    if ww or hh:
                        return sid, _stream_quality_label(ww, hh)
            except Exception:
                pass
        try:
            from .xtream_api import _get_session
            r = _get_session().get(url, timeout=8, allow_redirects=True)
            m = _re.search(r'RESOLUTION=(\d+)\x(\d+)', r.text, _re.IGNORECASE)
            if m:
                return sid, _stream_quality_label(int(m.group(1)), int(m.group(2)))
        except Exception:
            pass
        return sid, hint

    pw = Worker(_probe)
    pw.signals.result.connect(self._on_precache_result)
    pw.signals.error.connect(lambda _: None)
    self.pool.start(pw)

    def _on_precache_result(self, result):
        sid, label = result
        if label:
            self._live_q_cache[sid] = label
            self._save_quality_cache()
            # Update badge if this channel is currently visible
            for i in range(self.ch_list.count()):
                item = self.ch_list.item(i)
                if item:
                    s = item.data(Qt.ItemDataRole.UserRole)
                    if isinstance(s, dict) and str(s.get("stream_id", "")) == sid:
                        item.setData(QUALITY_ROLE, label)
                        self.ch_list.viewport().update()
                        break

    def _on_auth_fail(self, error: str):
        self._status("Connection failed")

```

```
        QMessageBox.critical(self, "Connection Error",
                              f"Could not connect:\n\n{error}")

    self._show_login()

def _go_home(self):
    if self._playing_stream is not None or self._ext_proc is not None:
        self._stop()
    self._stack.setCurrentIndex(0)

def _open_epg(self):
    if self._playing_stream is not None or self._ext_proc is not None:
        self._stop()
    self._epg_screen.open()
    self._stack.setCurrentIndex(2)

def _open_multiscreen(self):
    if self._playing_stream is not None or self._ext_proc is not None:
        self._stop()
    self._multi_screen.open()
    self._stack.setCurrentIndex(3)

def _open_catchup(self):
    self._auto_favorites = True
    self._go_to_section(0)

def _open_radio(self):
    if self._playing_stream is not None or self._ext_proc is not None:
        self._stop()
    if not self.api:
        return
    self._radio_screen.set_api(self.api)
    self._status("Loading radio stations...")
    w = Worker(lambda: self.api.get_live_streams(None))
    w.signals.result.connect(self._on_radio_streams_loaded)
    w.signals.error.connect(lambda e: self._status(f"Radio load error: {e}"))
    self.pool.start(w)

def _on_radio_streams_loaded(self, streams: list):
    radio = [s for s in streams if self._is_radio(s)]
    self._radio_streams_cache = radio
    if self.config.radio_favorite_ids:
        existing = {str(x.get("stream_id", "")) for x in self.config.radio_favorite_streams}
        fav_id_strs = {str(x) for x in self.config.radio_favorite_ids}
        added = False
        for s in radio:
            sid = str(s.get("stream_id", ""))
            if sid in fav_id_strs and sid not in existing:
                self.config.radio_favorite_streams.append(s)
                existing.add(sid)
                added = True
        if added:
            self.config.save()
    self._radio_screen.load(radio, "Radio Stations")
    self._stack.setCurrentIndex(5)

def _logoff(self):
    self.api = None
    self.config.account = Account()
    self.config.save()
    self.account_btn.setText("Login")
```

```
self._home.set_logged_out()
self._stack.setCurrentIndex(0)
self._show_login(force=True)

# — data loading —————

def _load_categories(self):
    self.cat_list.clear()
    self._show_spinner(self._cat_spinner)
    fn = [self.api.get_live_categories,
          self.api.get_vod_categories,
          self.api.get_series_categories,
          self.api.get_live_categories][self._tab]
    w = Worker(fn)
    w.signals.result.connect(self._on_categories)
    w.signals.error.connect(lambda e: self._status(f"Error: {e}"))
    self.pool.start(w)

def _on_categories(self, cats: list):
    self._all_cats = cats # cache for hidden-category name lookup
    self._hide_spinner(self._cat_spinner)
    self.cat_list.clear()

    # pinned non-draggable header items
    pinned = 0
    _no_drag = Qt.ItemFlag.ItemIsSelectable | Qt.ItemFlag.ItemIsEnabled | Qt.ItemFlag.ItemIsDropEnabled

    if self._tab == 3:
        for label, key, color in [
            (" FAVORITES", "__FAVORITES__", "#f9a825"),
            (" RECENT", "__RECENT__", "#42a5f5"),
            (" HIDDEN", "__HIDDEN__", "#c62828"),
        ]:
            item = QListWidgetItem(label)
            item.setData(Qt.ItemDataRole.UserRole, key)
            item.setForeground(QColor(color))
            item.setFlags(_no_drag)
            self.cat_list.addItem(item)
            pinned += 1
    elif self._tab in (0, 1, 2):
        for label, key, color in (
            [(" FAVORITES", "__FAVORITES__", "#f9a825"),
             (" RECENT", "__RECENT__", "#42a5f5")]
            if self._tab == 0 else
            [(" FAVORITES", "__FAVORITES__", "#f9a825"),
             (" CONTINUE WATCHING", "__CONTINUE__", "#42a5f5"),
             (" RECENTLY ADDED", "__RECENTLY_ADDED__", "#66bb6a")]
            if self._tab == 1 else
            [(" FAVORITES", "__FAVORITES__", "#f9a825"),
             (" CONTINUE WATCHING", "__CONTINUE__", "#42a5f5")]
        ):
            item = QListWidgetItem(label)
            item.setData(Qt.ItemDataRole.UserRole, key)
            item.setForeground(QColor(color))
            item.setFlags(_no_drag)
            self.cat_list.addItem(item)
            pinned += 1

    all_item = QListWidgetItem("ALL")
    all_item.setData(Qt.ItemDataRole.UserRole, None)
```

```
all_item.setFlags(_no_drag)
self.cat_list.addItem(all_item)
pinned += 1

if self._tab in (0, 3):
    self.known_live_cat_ids = {
        str(c.get("category_id", ""))
        for c in cats
        if c.get("category_id") and str(c.get("category_id")) != "0"
    }

self.cat_list.pinned_rows = pinned

# apply saved order
saved = self.config.cat_order.get(str(self._tab), [])
if saved:
    order_map = {str(cid): i for i, cid in enumerate(saved)}
    cats = sorted(cats, key=lambda c: order_map.get(str(c.get("category_id", "")), len(saved)))

if self._tab != 3:
    hidden_cats = {str(x) for x in self.config.hidden_category_ids}
    for c in cats:
        cid = c.get("category_id")
        if str(cid) in hidden_cats:
            continue
        name = c.get("category_name", "-")
        if self._tab == 2 and name.strip().upper() in ("ALL", "ALL SERIES", "ALL SERIE"):
            continue
        item = QListWidgetItem(name.upper())
        item.setData(Qt.ItemDataRole.UserRole, cid)
        self.cat_list.addItem(item)

if self._tab == 0:
    unc_item = QListWidgetItem(" UNCATEGORIZED")
    unc_item.setData(Qt.ItemDataRole.UserRole, "__UNCATEGORIZED__")
    unc_item.setFlags(_no_drag)
    self.cat_list.addItem(unc_item)

if self._tab == 0:
    hid_item = QListWidgetItem(" HIDDEN")
    hid_item.setData(Qt.ItemDataRole.UserRole, "__HIDDEN__")
    hid_item.setForeground(QColor("#c62828"))
    hid_item.setFlags(_no_drag)
    self.cat_list.addItem(hid_item)

has_favs = False
if self._tab == 0: # Live TV – exclude movies, series, and radio
    has_favs = any(
        not self._is_radio(s) and not s.get("container_extension") and not s.get("series_id")
        for s in self.config.favorite_streams
    )
elif self._tab == 1: # Movies – only VOD favorites count
    has_favs = any(s.get("container_extension") for s in self.config.favorite_streams)
elif self._tab == 2: # Series – only series favorites count
    has_favs = any(s.get("series_id") for s in self.config.favorite_streams)
elif self._tab == 3: # Radio – only actual radio favorites count
    has_favs = (
        any(self._is_radio(s) for s in self.config.favorite_streams)
        or bool(self.config.radio_favorite_ids)
    )
)
```

```

if has_favs:
    # Verify the filtered list will actually have items before defaulting to FAVORITES
    if self._tab == 3:
        real_favs = self.config.radio_favorite_streams
    elif self._tab == 0:
        real_favs = [s for s in self.config.favorite_streams
                      if not self._is_radio(s)
                      and not s.get("container_extension")
                      and not s.get("series_id")]
        if self._known_live_cat_ids:
            real_favs = [s for s in real_favs
                          if not s.get("category_id")
                          or str(s.get("category_id")) in ("0", "")
                          or str(s.get("category_id")) in self._known_live_cat_ids]
    else:
        real_favs = self.config.favorite_streams
    has_favs = bool(real_favs)

if has_favs:
    self.cat_list.setCurrentRow(0) # FAVORITES
    if self._tab == 3:
        self._show_saved_streams(self.config.radio_favorite_streams, "FAVORITES", raw=True)
    else:
        self._show_saved_streams(self.config.favorite_streams, "FAVORITES")
elif self._tab == 0:
    # Default to Canadian category if available, else ALL
    canada_row, canada_cid = None, None
    for i in range(self.cat_list.count()):
        item = self.cat_list.item(i)
        if item:
            cid = item.data(Qt.ItemDataRole.UserRole)
            if cid and not str(cid).startswith("__") and "CANAD" in item.text().upper():
                canada_row, canada_cid = i, cid
                break
    if canada_row is not None:
        self.cat_list.setCurrentRow(canada_row)
        self._load_channels(category_id=canada_cid)
    else:
        self.cat_list.setCurrentRow(pinned - 1) # ALL
        self._load_channels(category_id=None)
else:
    self.cat_list.setCurrentRow(pinned - 1) # ALL
    self._load_channels(category_id=None)

def _on_category_clicked(self, item: QListWidgetItem):
    # If a movie/series is playing, save progress then restore the grid layout
    if self._tab in (1, 2) and self._playing_stream is not None:
        self._save_progress()
        self._back_to_browse()

    if self._cat_select_mode:
        cid = item.data(Qt.ItemDataRole.UserRole)
        if cid and not str(cid).startswith("__"):
            new_state = (Qt.CheckState.Unchecked
                        if item.checkState() == Qt.CheckState.Checked
                        else Qt.CheckState.Checked)
            item.setCheckState(new_state)
            self._update_cat_select_bar()
        return
    if self._ch_select_mode:

```

```

        self._exit_ch_select_mode()
    self.search_edit.clear()
    key = item.data(Qt.ItemDataRole.UserRole)
    if key == "__FAVORITES__":
        self._filter_uncategorized = False
        self._filter_hidden = False
        if self._tab == 3:
            self._show_saved_streams(self.config.radio_favorite_streams, "FAVORITES", raw=True)
        else:
            self._show_saved_streams(self.config.favorite_streams, "FAVORITES")
    elif key == "__RECENT__":
        self._filter_uncategorized = False
        self._filter_hidden = False
        self._show_saved_streams(self.config.recent_streams, "RECENT")
    elif key == "__CONTINUE__":
        self._filter_uncategorized = False
        self._filter_hidden = False
        self._show_continue_watching()
    elif key == "__RECENTLY_ADDED__":
        self._filter_uncategorized = False
        self._filter_hidden = False
        self._load_channels(category_id="__RECENTLY_ADDED__")
    else:
        self._load_channels(category_id=key)

def _show_saved_streams(self, streams: list, label: str, raw: bool = False):
    self._img_queue.clear()
    self._epg_queue.clear()
    self._season_queue.clear()
    self._vod_q_queue.clear()
    self._live_q_queue.clear()
    self._load_gen += 1
    gen = self._load_gen
    if self._tab in (1, 2):
        # poster grid view for movies / series – filter to correct content type
        fav_set = {_fav_id(x) for x in self.config.favorite_streams}
        live_cat_ids = self._known_live_cat_ids or set()
        if self._tab == 1: # Movies: VOD streams have container_extension, live streams don't
            streams = [s for s in streams if _fav_id(s) in fav_set
                        and s.get("container_extension")
                        and not self._is_radio(s)
                        and str(s.get("category_id", "")) not in live_cat_ids]
        else: # Series: have series_id (and a valid id)
            streams = [s for s in streams if _fav_id(s) in fav_set
                        and s.get("series_id")
                        and str(s.get("series_id", "")) not in live_cat_ids]
    self.poster_grid.clear()
    if not streams:
        item = QListWidgetItem("No favorites set")
        item.setFlags(Qt.ItemFlag.NoItemFlags)
        item.setSizeHint(QSize(self.poster_grid.width() or 600, 120))
        self.poster_grid.addItem(item)
        return
    self._ch_header.setText(f"{label} ({len(streams)})")
    blank = QIcon(self._blank_poster())
    self.poster_grid.setUpdatesEnabled(False)
    disk_cached: set = set()
    for i, s in enumerate(streams):
        item = QListWidgetItem("")
        item.setData(Qt.ItemDataRole.UserRole, s)

```

```

        item.setData(FAV_ROLE, True)
        sid = str(s.get("stream_id", "") or s.get("series_id", ""))
        cached_path = POSTER_CACHE_DIR / f"{sid}.jpg" if sid else None
        if cached_path and cached_path.exists():
            px = QPixmap(str(cached_path))
            if not px.isNull():
                item.setData(BASE_POSTER_ROLE, px)
                item.setIcon(QIcon(px))
                disk_cached.add(i)
            else:
                item.setIcon(blank)
        else:
            url = s.get("stream_icon") or s.get("cover") or ""
            if url:
                item.setIcon(blank)
            else:
                gp = self._make_generic_poster(s.get("name", ""))
                item.setData(BASE_POSTER_ROLE, gp)
                item.setIcon(QIcon(gp))
            self.poster_grid.addItem(item)
        self.poster_grid.setUpdatesEnabled(True)
        # Render badges on disk-cached posters immediately
        for i in disk_cached:
            it = self.poster_grid.item(i)
            if it:
                self._refresh_poster_icon(it, gen)
        # Only download posters not already on disk
        for i, s in enumerate(streams):
            if i not in disk_cached:
                url = s.get("stream_icon") or s.get("cover") or ""
                if url:
                    self._fetch_img(i, url, gen, _POSTER_W, _POSTER_H, s.get("name", ""), "")
        self._flush_img_queue()
        if self._tab == 1:
            self._vod_q_queue = [
                (i, str(s.get("stream_id", "")))
                for i, s in enumerate(streams)
                if s.get("stream_id")
            ]
            self._flush_vod_q_queue(gen)
        elif self._tab == 2:
            self._season_queue = [
                (i, str(s.get("series_id", "")))
                for i, s in enumerate(streams)
                if s.get("series_id")
            ]
            self._flush_season_queue(gen)
        return

self.ch_list.clear()
if not raw:
    if self._tab == 0:
        streams = [s for s in streams
                    if not self._is_radio(s)
                    and not s.get("container_extension")
                    and not s.get("series_id")]
        if self._known_live_cat_ids:
            streams = [s for s in streams
                      if not s.get("category_id")
                      or str(s.get("category_id")) in ("0", "")

```



```

        or str(s.get("category_id")) in self._known_live_cat_ids]
    elif self._tab == 3:
        streams = [s for s in streams if self._is_radio(s)]
    if not streams:
        _placeholder(self.ch_list, f"No {label.lower()} yet")
        return
    fav_ids = (
        {x.get("stream_id") for x in self.config.radio_favorite_streams}
        if self._tab == 3
        else {x.get("stream_id") for x in self.config.favorite_streams}
    )
    self.ch_header.setText(f"{label} ({len(streams)})")
    self.ch_list.setUpdatesEnabled(False)
    for s in streams:
        name = s.get("name", "-")
        dname = self._dn(name)
        sid = str(s.get("stream_id", ""))
        item = QListWidgetItem(dname)
        item.setData(Qt.ItemDataRole.UserRole, s)
        item.setData(EPG_ROLE, "")
        item.setData(EPG_PROG_ROLE, None)
        item.setData(RADIO_ROLE, self._is_radio(s))
        item.setData(FAV_ROLE, s.get("stream_id") in fav_ids)
        item.setData(FAIL_ROLE, self.config.is_stream_unreliable(sid))
        _is_radio = self._is_radio(s)
        if not _is_radio and self._tab != 3:
            item.setData(QUALITY_ROLE, _stream_quality_hint(s, self._live_q_cache.get(sid)))
            item.setIcon(QIcon(_make_radio_avatar(_LOGO_SZ) if _is_radio else _make_avatar(dname, _LOGO_SZ)))
        self.ch_list.addItem(item)
    self.ch_list.setUpdatesEnabled(True)
    for i, s in enumerate(streams):
        url = s.get("stream_icon", "")
        if url:
            self._fetch_img(i, url, gen, _LOGO_SZ, _LOGO_SZ, "", "")
    self._flush_img_queue()
    self._epg_queue = [
        (i, s.get("stream_id"))
        for i, s in enumerate(streams)
        if s.get("stream_id")
    ]
    self._flush_epg_queue(gen)
    # queue live quality fetches for Live TV only (not radio)
    if self._tab == 0:
        self._live_q_queue = [
            (i, str(s.get("stream_id", "")), s.get("name", ""))
            for i, s in enumerate(streams)
            if s.get("stream_id")
            and str(s.get("stream_id", "")) not in self._live_q_cache
        ]
        self._flush_live_q_queue(gen)

def _cat_context_menu(self, pos):
    item = self.cat_list.itemAt(pos)
    if not item:
        return
    cid = item.data(Qt.ItemDataRole.UserRole)

    if str(cid) == "__FAVORITES__":
        menu = QMenu(self)
        menu.setStyleSheet(

```

```

        "QMenu{background:#0d1117;color:#d0d8e8;border:1px solid #1e2a3a;}"
        "QMenu::item{padding:8px 20px;}"
        "QMenu::item:selected{background:#c62828;}"
    )
    act = menu.addAction("  Clear all favorites")
    act.triggered.connect(self._clear_all_favorites)
    menu.exec(self.cat_list.mapToGlobal(pos))
    return

# only real categories (not virtual pinned items)
if not cid or str(cid).startswith("__"):
    return
is_hidden = str(cid) in {str(x) for x in self.config.hidden_category_ids}
menu = QMenu(self)
menu.setStyleSheet(
    "QMenu{background:#0d1117;color:#d0d8e8;border:1px solid #1e2a3a;}"
    "QMenu::item{padding:8px 20px;}"
    "QMenu::item:selected{background:#1565c0;}"
)
act = menu.addAction("  Unhide category" if is_hidden else "  Hide category")
act.triggered.connect(lambda: self._toggle_hidden_category(cid, not is_hidden))
menu.exec(self.cat_list.mapToGlobal(pos))

def _clear_all_favorites(self):
    if self._tab == 3:
        count = len(self.config.radio_favorite_streams)
        kind = "radio station"
    elif self._tab == 2:
        count = sum(1 for s in self.config.favorite_streams if s.get("series_id"))
        kind = "series"
    elif self._tab == 1:
        count = sum(1 for s in self.config.favorite_streams if s.get("container_extension"))
        kind = "movie"
    else:
        count = sum(1 for s in self.config.favorite_streams
                     if not self._is_radio(s) and not s.get("container_extension")
                     and not s.get("series_id"))
        kind = "channel"

    if count == 0:
        return

    plural = "s" if count != 1 else ""
    reply = QMessageBox.question(
        self, "Clear Favorites",
        f"Remove all {count} {kind}{plural} from favorites?",
        QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.Cancel,
        QMessageBox.StandardButton.Cancel,
    )
    if reply != QMessageBox.StandardButton.Yes:
        return

    if self._tab == 3:
        for s in self.config.radio_favorite_streams:
            self._delete_fav_poster(s)
        self.config.radio_favorite_streams.clear()
        self.config.radio_favorite_ids.clear()
    elif self._tab in (1, 2):
        keep, remove = [], []
        for s in self.config.favorite_streams:

```

```

        if (self._tab == 1 and s.get("container_extension")) or \
            (self._tab == 2 and s.get("series_id")):
            remove.append(s)
        else:
            keep.append(s)
    for s in remove:
        self._delete_fav_poster(s)
    self.config.favorite_streams = keep
else:
    keep, remove = [], []
    for s in self.config.favorite_streams:
        if not self._is_radio(s) and not s.get("container_extension") \
            and not s.get("series_id"):
            remove.append(s)
        else:
            keep.append(s)
    self.config.favorite_streams = keep

self.config.save()
QTimer.singleShot(0, self._load_categories)

def _toggle_hidden_category(self, cid, hide: bool):
    cid_str = str(cid)
    self.config.hidden_category_ids = [
        x for x in self.config.hidden_category_ids if str(x) != cid_str
    ]
    if hide:
        self.config.hidden_category_ids.append(cid_str)
    self.config.save()
    QTimer.singleShot(0, self._load_categories)

# — multi-select hide —————

def _toggle_cat_select_mode(self):
    if self._cat_select_btn.isChecked():
        self._cat_select_mode = True
        self.cat_list.setDragDropMode(QListWidget.DragDropMode.NoDragDrop)
        pinned = self.cat_list.pinned_rows
        for i in range(pinned, self.cat_list.count()):
            item = self.cat_list.item(i)
            if item:
                cid = item.data(Qt.ItemDataRole.UserRole)
                if cid and not str(cid).startswith("__"):
                    item.setFlags(item.flags() | Qt.ItemFlag.ItemIsUserCheckable)
                    item.setCheckState(Qt.CheckState.Unchecked)
        self._cat_action_bar.show()
        self._update_cat_select_bar()
    else:
        self._exit_cat_select_mode()

def _exit_cat_select_mode(self):
    self._cat_select_mode = False
    self._cat_select_btn.setChecked(False)
    for i in range(self.cat_list.count()):
        item = self.cat_list.item(i)
        if item:
            item.setFlags(item.flags() & ~Qt.ItemFlag.ItemIsUserCheckable)
    self._cat_action_bar.hide()
    self.cat_list.setDragDropMode(QListWidget.DragDropMode.InternalMove)

```

```

def _update_cat_select_bar(self):
    count = sum(
        1 for i in range(self.cat_list.count())
        if (item := self.cat_list.item(i)) and item.checkState() == Qt.CheckState.Checked
    )
    self._cat_hide_btn.setText(f"Hide {count} Selected")
    self._cat_hide_btn.setEnabled(count > 0)

def _hide_selected_categories(self):
    hidden = {str(x) for x in self.config.hidden_category_ids}
    for i in range(self.cat_list.count()):
        item = self.cat_list.item(i)
        if item and item.checkState() == Qt.CheckState.Checked:
            cid = str(item.data(Qt.ItemDataRole.UserRole))
            if cid not in hidden:
                self.config.hidden_category_ids.append(cid)
    self.config.save()
    self._exit_cat_select_mode()
    QTimer.singleShot(0, self._load_categories)

# — channel multi-select hide —————

def _toggle_ch_select_mode(self):
    if self._ch_select_btn.isChecked():
        self._ch_select_mode = True
        self._ch_delegate.select_mode = True
        for i in range(self.ch_list.count()):
            item = self.ch_list.item(i)
            if item:
                s = item.data(Qt.ItemDataRole.UserRole)
                if s and not s.get("__hidden_cat__") and s.get("stream_id"):
                    item.setCheckState(Qt.CheckState.Unchecked)
        self.ch_list.viewport().update()
        self._ch_action_bar.show()
        self._update_ch_select_bar()
    else:
        self._exit_ch_select_mode()

def _exit_ch_select_mode(self):
    self._ch_select_mode = False
    self._ch_select_btn.setChecked(False)
    self._ch_delegate.select_mode = False
    self.ch_list.viewport().update()
    self._ch_action_bar.hide()

def _update_ch_select_bar(self):
    count = sum(
        1 for i in range(self.ch_list.count())
        if (item := self.ch_list.item(i)) and item.checkState() == Qt.CheckState.Checked
    )
    self._ch_hide_btn.setText(f"Hide {count} Selected")
    self._ch_hide_btn.setEnabled(count > 0)

def _hide_selected_channels(self):
    hidden = {str(x) for x in self.config.hidden_stream_ids}
    for i in range(self.ch_list.count()):
        item = self.ch_list.item(i)
        if item and item.checkState() == Qt.CheckState.Checked:
            s = item.data(Qt.ItemDataRole.UserRole)
            if s:

```

```

        sid = str(s.get("stream_id", ""))
        if sid and sid not in hidden:
            self.config.hidden_stream_ids.append(sid)
self.config.save()
self._exit_ch_select_mode()
cat_id = (self.cat_list.currentItem().data(Qt.ItemDataRole.UserRole)
          if self.cat_list.currentItem() else None)
QTimer.singleShot(0, lambda: self._load_channels(category_id=cat_id))

def _ch_context_menu(self, pos):
    item = self.ch_list.itemAt(pos)
    if not item:
        return
    s = item.data(Qt.ItemDataRole.UserRole)
    if not isinstance(s, dict):
        return
    menu = QMenu(self)
    menu.setStyleSheet(
        "QMenu{background:#0d1117;color:#d0d8e8;border:1px solid #1e2a3a;}"
        "QMenu::item{padding:8px 20px;}"
        "QMenu::item:selected{background:#1565c0;}"
    )
    # hidden category header row
    if s.get("__hidden_cat__"):
        cid = s.get("category_id")
        cname = s.get("category_name", "")
        act = menu.addAction(f"  Unhide category: {cname.upper()}")
        act.triggered.connect(lambda: self._toggle_hidden_category(cid, False))
        menu.exec(self.ch_list.mapToGlobal(pos))
        return
    sid = s.get("stream_id")
    if self._tab == 3:
        is_fav = any(x.get("stream_id") == sid for x in self.config.radio_favorite_streams)
    else:
        is_fav = any(x.get("stream_id") == sid for x in self.config.favorite_streams)
    if is_fav:
        act = menu.addAction("★ Remove from Favorites")
    else:
        act = menu.addAction("✧ Add to Favorites")
    act.triggered.connect(lambda: self._toggle_favorite(s, not is_fav))

    is_hidden = str(sid) in {str(x) for x in self.config.hidden_stream_ids}
    hide_act = menu.addAction("  Unhide channel" if is_hidden else "  Hide channel")
    hide_act.triggered.connect(lambda: self._toggle_hidden(s, not is_hidden))

    menu.exec(self.ch_list.mapToGlobal(pos))

_VIRTUAL_CAT_KEYS = {None, "__FAVORITES__", "__RECENT__", "__CONTINUE__",
                     "__UNCATEGORIZED__", "__HIDDEN__"}

def _on_cat_reordered(self):
    order = []
    for i in range(self.cat_list.count()):
        item = self.cat_list.item(i)
        cid = item.data(Qt.ItemDataRole.UserRole)
        if cid not in self._VIRTUAL_CAT_KEYS:
            order.append(str(cid))
    self.config.cat_order[str(self._tab)] = order
    self.config.save()

```

```
def _toggle_favorite(self, s: dict, add: bool):
    sid = s.get("stream_id")
    if self._tab == 3:
        self.config.radio_favorite_streams = [
            x for x in self.config.radio_favorite_streams if x.get("stream_id") != sid
        ]
        self.config.radio_favorite_ids = [
            x for x in self.config.radio_favorite_ids if x != sid
        ]
        if add:
            self.config.radio_favorite_streams.append(s)
            self.config.radio_favorite_ids.append(sid)
    else:
        self.config.favorite_streams = [
            x for x in self.config.favorite_streams if x.get("stream_id") != sid
        ]
        if add:
            self.config.favorite_streams.append(s)
    self.config.save()

def _toggle_hidden(self, s: dict, hide: bool):
    sid = str(s.get("stream_id", ""))
    self.config.hidden_stream_ids = [
        x for x in self.config.hidden_stream_ids if str(x) != sid
    ]
    if hide:
        self.config.hidden_stream_ids.append(sid)
    self.config.save()
    cat_id = (self.cat_list.currentItem().data(Qt.ItemDataRole.UserRole)
              if self.cat_list.currentItem() else None)
    QTimer.singleShot(0, lambda: self._load_channels(category_id=cat_id))

def _on_resolution(self, label: str):
    self.player.hide_message() # a video frame has arrived
    # Normalise player labels to our standard badge set
    _norm = {"480p": "SD", "360p": "SD", "240p": "SD"}
    label = _norm.get(label, label)

    if label:
        self._player_badge.setText(label)
        self._player_badge.show()
    else:
        self._player_badge.hide()

    # Save to cache and update the badge on the matching row
    if label and self._playing_stream:
        sid = str(self._playing_stream.get("stream_id", ""))
        if sid:
            if self._tab in (1, 2):
                # VOD or Series - update VOD quality cache and poster grid
                self._vod_q_cache[sid] = label
                for i in range(self.poster_grid.count()):
                    item = self.poster_grid.item(i)
                    if item:
                        s = item.data(Qt.ItemDataRole.UserRole)
                        if isinstance(s, dict) and str(s.get("stream_id", "")) == sid:
                            item.setData(VOD_QUALITY_ROLE, label)
                            self.poster_grid.viewport().update()
                            break
            else:
```

```

        # Live TV or Radio – update live quality cache and channel list
        self._live_q_cache[sid] = label
        self._save_quality_cache()
        # Find the item by stream_id – not currentItem() which may differ
        for i in range(self.ch_list.count()):
            item = self.ch_list.item(i)
            if item:
                s = item.data(Qt.ItemDataRole.UserRole)
                if isinstance(s, dict) and str(s.get("stream_id", "")) == sid:
                    item.setData(QUALITY_ROLE, label)
                    self.ch_list.viewport().update()
                    break

def _ch_prev(self):
    if self._tab not in (0, 3) or not self._playing_stream:
        return
    row = self.ch_list.currentRow()
    if row > 0:
        self.ch_list.setCurrentRow(row - 1)
        self._on_channel_activated(self.ch_list.currentItem())

def _ch_next(self):
    if self._tab not in (0, 3) or not self._playing_stream:
        return
    row = self.ch_list.currentRow()
    if row < self.ch_list.count() - 1:
        self.ch_list.setCurrentRow(row + 1)
        self._on_channel_activated(self.ch_list.currentItem())

def _load_channels(self, category_id=None):
    if self._tab in (0, 3):
        self.ch_list.clear()
        self._show_spinner(self._ch_spinner)
    else:
        self.poster_grid.clear()
        self._show_spinner(self._grid_spinner)

    self._img_queue.clear()
    self._epg_queue.clear()
    self._season_queue.clear()
    self._vod_q_queue.clear()
    self._live_q_queue.clear()
    self._probe_status_lbl.setVisible(False)
    self._load_gen += 1
    gen = self._load_gen # capture now; guard against stale callbacks

    # UNCATEGORIZED and HIDDEN and WORKING are virtual categories – load all streams, filter later
    self._filter_uncategorized = (category_id == "__UNCATEGORIZED__")
    self._filter_hidden = (category_id == "__HIDDEN__")
    self._filter_working = (category_id == "__WORKING__")
    self._filter_recently_added = (category_id == "__RECENTLY_ADDED__")
    api_cat = None if (self._filter_uncategorized or self._filter_hidden or self._filter_working or self._filter_recently_added) else category_id

    # Radio ALL view: skip the API call and reuse the cached stream list
    if self._tab == 3 and not self._filter_hidden and not self._filter_working and category_id is None and self._radio_all_streams:
        QTimer.singleShot(0, lambda g=gen: self._on_channels(list(self._radio_all_streams), g))
    return

```

```

fn = [lambda: self.api.get_live_streams(api_cat),
      lambda: self.api.get_vod_streams(api_cat),
      lambda: self.api.get_series(api_cat),
      lambda: self.api.get_live_streams(api_cat)][self._tab]
w = Worker(fn)
w.signals.result.connect(lambda streams, g=gen: self._on_channels(streams, g))
w.signals.error.connect(lambda e: self._status(f"Error: {e}"))
self.pool.start(w)

def _on_channels(self, streams: list, gen: int):
    if gen != self._load_gen:
        return # stale result; a newer load is already in progress
    self._hide_spinner(self._ch_spinner)
    self._hide_spinner(self._grid_spinner)
    labels = ["CHANNELS", "MOVIES", "SERIES", "RADIO"]

    if self._tab == 3 and self.config.radio_favorite_ids:
        existing = {str(x.get("stream_id", "")) for x in self.config.radio_favorite_streams}
        fav_id_strs = {str(x) for x in self.config.radio_favorite_ids}
        added = False
        for s in streams:
            sid = str(s.get("stream_id", ""))
            if sid in fav_id_strs and sid not in existing:
                self.config.radio_favorite_streams.append(s)
                existing.add(sid)
                added = True
        if added:
            self.config.save()

    hidden_sids = {str(x) for x in self.config.hidden_stream_ids}
    hidden_cids = {str(x) for x in self.config.hidden_category_ids}
    fav_ids = (
        {x.get("stream_id") for x in self.config.radio_favorite_streams}
        if self._tab == 3
        else {x.get("stream_id") for x in self.config.favorite_streams}
    )

    def _is_hidden(s):
        return (str(s.get("stream_id", "")) in hidden_sids or
                str(s.get("category_id", "")) in hidden_cids)

    if self._tab == 0:
        streams = [s for s in streams if not self._is_radio(s)]
        if self._filter_hidden:
            streams = [s for s in streams if str(s.get("stream_id", "")) in hidden_sids]
        elif self._filter_uncategorized:
            streams = [
                s for s in streams
                if str(s.get("category_id", "0")) not in self._known_live_cat_ids
                and not _is_hidden(s)
            ]
        else:
            streams = [s for s in streams if not _is_hidden(s)]
            if self._known_live_cat_ids:
                streams = [s for s in streams
                           if not s.get("category_id")
                           or str(s.get("category_id")) in ("0", "")
                           or str(s.get("category_id")) in self._known_live_cat_ids]
            streams = sorted(streams, key=lambda s: (0 if s.get("stream_id") in fav_ids else 1,
                                                       (s.get("name") or "").lower()))

```



```

elif self._tab == 3:
    if self._filter_working:
        working_sids = {str(x) for x in self.config.working_radio_ids}
        streams = [s for s in streams if str(s.get("stream_id", "")) in working_sids]
    else:
        streams = [
            s for s in streams
            if str(s.get("category_id", "0")) not in self._known_live_cat_ids
        ]
        if not self._filter_hidden and not self._radio_all_streams:
            self._radio_all_streams = list(streams) # populate cache before hidden filter
        if self._filter_hidden:
            streams = [s for s in streams if _is_hidden(s) and self._is_radio(s)]
        else:
            streams = [s for s in streams if not _is_hidden(s)]
        streams = sorted(streams, key=lambda s: (0 if s.get("stream_id") in fav_ids else 1,
                                                (s.get("name") or "").lower()))
        self._radio_streams_cache = streams

if self._tab in (0, 3):
    self.ch_list.clear()
    self._ch_header.setText(f"{{labels[0]}} ({{len(streams)}})")
    self.ch_list.setUpdatesEnabled(False)

# when viewing HIDDEN, show hidden category names as header entries
if self._filter_hidden and self.config.hidden_category_ids:
    cat_name_map = {str(c.get("category_id", "")): c.get("category_name", "-")
                    for c in self._all_cats}

    # for radio tab, only show headers for categories that have hidden radio stations
    stream_cat_ids = {str(s.get("category_id", "")) for s in streams} if self._tab == 3 else None
    for cid in self.config.hidden_category_ids:
        if stream_cat_ids is not None and str(cid) not in stream_cat_ids:
            continue
        cname = cat_name_map.get(str(cid), f"Category {cid}")
        hdr = QListWidgetItem(f" {{cname.upper()}}")
        hdr.setData(Qt.ItemDataRole.UserRole, {"__hidden_cat__": True, "category_id": cid, "category_name": cn>
ame}})

        hdr.setForeground(QColor("#c62828"))
        hdr.setFlags(Qt.ItemFlag.ItemIsSelectable | Qt.ItemFlag.ItemIsEnabled)
        self.ch_list.addItem(hdr)

for s in streams:
    name = s.get("name", "-")
    dname = self._dn(name)
    item = QListWidgetItem(dname)
    item.setData(Qt.ItemDataRole.UserRole, s)
    item.setData(EPG_ROLE, "")
    item.setData(RADIO_ROLE, self._is_radio(s))
    item.setData(FAV_ROLE, s.get("stream_id") in fav_ids)
    sid = str(s.get("stream_id", ""))
    item.setData(FAIL_ROLE, self.config.is_stream_unreliable(sid))
    _is_radio = self._is_radio(s)
    if not _is_radio and self._tab != 3:
        item.setData(QUALITY_ROLE, _stream_quality_hint(s, self._live_q_cache.get(sid)))
        item.setIcon(QIcon(_make_radio_avatar(_LOGO_SZ) if _is_radio else _make_avatar(dname, _LOGO_SZ)))
        self.ch_list.addItem(item)
    self.ch_list.setUpdatesEnabled(True)
for i, s in enumerate(streams):
    url = s.get("stream_icon", "")
    if url:

```

```

        self._fetch_img(i, url, gen, _LOGO_SZ, _LOGO_SZ, "", "")
    self._flush_img_queue()
    # queue EPG subtitle fetch for every channel
    self._epg_queue = [
        (i, s.get("stream_id"))
        for i, s in enumerate(streams)
        if s.get("stream_id")
    ]
    self._flush_epg_queue(gen)
    # queue live quality fetches for ALL live TV streams
    if self._tab == 0:
        self._live_q_queue = [
            (i, str(s.get("stream_id", "")), s.get("name", ""))
            for i, s in enumerate(streams)
            if s.get("stream_id")
            and str(s.get("stream_id", "")) not in self._live_q_cache
        ]
        self._flush_live_q_queue(gen)
    else:
        fav_set = {_fav_id(x) for x in self.config.favorite_streams}
        if self._filter_recently_added:
            cutoff = time.time() - 30 * 86400
            streams = [s for s in streams
                        if int(s.get("added", 0) or 0) >= cutoff]
            streams = sorted(streams, key=lambda s: -int(s.get("added", 0) or 0))
        else:
            streams = sorted(streams, key=lambda s: (0 if _fav_id(s) in fav_set else 1,
                                                    (s.get("name") or "").lower()))

        self.poster_grid.clear()
        blank_icon = QIcon(self._blank_poster())
        self.poster_grid.setUpdatesEnabled(False)
        for s in streams:
            item = QListWidgetItem("")
            item.setData(Qt.ItemDataRole.UserRole, s)
            item.setData(FAV_ROLE, _fav_id(s) in fav_set)
            url = s.get("stream_icon") or s.get("cover") or ""
            if url:
                item.setIcon(blank_icon)
            else:
                gp = self._make_generic_poster(s.get("name", ""))
                item.setData(BASE_POSTER_ROLE, gp)
                item.setIcon(QIcon(gp))
            self.poster_grid.addItem(item)
        self.poster_grid.setUpdatesEnabled(True)
        for i, s in enumerate(streams):
            url = s.get("stream_icon") or s.get("cover") or ""
            if url:
                self._fetch_img(i, url, gen, _POSTER_W, _POSTER_H,
                                s.get("name", ""),
                                str(s.get("rating") or ""))
        self._flush_img_queue()
        if self._tab == 1:
            self._vod_q_queue = [
                (i, str(s.get("stream_id", "")))
                for i, s in enumerate(streams)
                if s.get("stream_id")
            ]
            self._flush_vod_q_queue(gen)
        elif self._tab == 2:
            self._season_queue = [

```

```

        (i, str(s.get("series_id", "")))
        for i, s in enumerate(streams)
        if s.get("series_id")
    ]
    self._flush_season_queue(gen)

    self._status(f"Loaded {len(streams)} items")

# — image loading —————

def _blank_poster(self) -> QPixmap:
    px = QPixmap(_POSTER_W, _POSTER_H)
    px.fill(QColor("#111827"))
    return px

def _make_generic_poster(self, name: str) -> QPixmap:
    px = QPixmap(_POSTER_W, _POSTER_H)
    px.fill(QColor("#0d1117"))
    p = QPainter(px)
    p.setRenderHint(QPainter.RenderHint.Antialiasing)
    p.setRenderHint(QPainter.RenderHint.TextAntialiasing)

    # background gradient
    grad = QLinearGradient(0, 0, 0, _POSTER_H)
    grad.setColorAt(0.0, QColor("#0d1b2a"))
    grad.setColorAt(1.0, QColor("#0a0f1a"))
    p.fillRect(0, 0, _POSTER_W, _POSTER_H, grad)

    # border
    p.setPen(QPen(QColor("#1e2a3a"), 1))
    p.setBrush(Qt.BrushStyle.NoBrush)
    p.drawRect(1, 1, _POSTER_W - 2, _POSTER_H - 2)

    # film-strip notches at top and bottom
    p.setPen(Qt.PenStyle.NoPen)
    p.setBrush(QColor("#1e2a3a"))
    for x in range(4, _POSTER_W - 4, 14):
        p.drawRoundedRect(x, 5, 9, 6, 2, 2)
        p.drawRoundedRect(x, _POSTER_H - 11, 9, 6, 2, 2)

    # movie title – centred, word-wrapped
    p.setPen(QColor("#d0d8e8"))
    fn = QFont(); fn.setPixelSize(12); fn.setBold(True)
    p.setFont(fn)
    fm = p.fontMetrics()
    pad = 10
    max_w = _POSTER_W - pad * 2
    words = name.split()
    lines, cur = [], ""
    for w in words:
        test = (cur + " " + w).strip()
        if fm.horizontalAdvance(test) <= max_w:
            cur = test
        else:
            if cur:
                lines.append(cur)
            cur = w
    if cur:
        lines.append(cur)
    lines = lines[:4] # max 4 lines

```

```

        line_h = fm.height() + 2
        total_h = len(lines) * line_h
        y = (_POSTER_H - total_h) // 2
        for line in lines:
            x = (_POSTER_W - fm.horizontalAdvance(line)) // 2
            p.drawText(x, y + fm.ascent(), line)
            y += line_h

        # subtle ► icon below title
        p.setPen(QColor("#1e3a5a"))
        fi = QFont(); fi.setPixelSize(20)
        p.setFont(fi)
        play_y = (_POSTER_H - total_h) // 2 + total_h + 10
        if play_y + 24 < _POSTER_H - 14:
            p.drawText(0, play_y, _POSTER_W, 24, Qt.AlignmentFlag.AlignCenter, "►")

        p.end()
        return px

def _fetch_img(self, row, url, gen, tw, th, name, rating):
    key = f"{url}_{tw}x{th}"
    if key in self._img_cache:
        self._apply_img(row, self._img_cache[key], gen)
        return
    self._img_queue.append((row, url, gen, tw, th, name, rating))

def _flush_img_queue(self):
    while self._img_queue and self._img_active < 4:
        self._start_next_img()

def _start_next_img(self):
    if not self._img_queue:
        return
    row, url, gen, tw, th, name, rating = self._img_queue.pop(0)
    self._img_active += 1
    key = f"{url}_{tw}x{th}"

    def dl(u=url):
        import urllib.request as _urllib
        req = _urllib.Request(u, headers={"User-Agent": "Mozilla/5.0"})
        with _urllib.urlopen(req, timeout=8) as resp:
            return resp.read() # return raw bytes only – no Qt objects in threads

    w = Worker(dl)
    w.signals.result.connect(
        lambda data, r=row, g=gen, k=key, tw_=tw, th_=th, n=name, rt=rating:
            self._on_img_done(r, data, g, k, tw_, th_, n, rt)
    )
    w.signals.error.connect(lambda _: self._on_img_fetch_error())
    self.pool.start(w)

def _on_img_done(self, row, data, gen, key, tw, th, name, rating):
    self._img_active -= 1
    img = QImage()
    img.loadFromData(data)
    pixmap = QPixmap.fromImage(img)
    self._on_img(row, pixmap, gen, key, tw, th, name, rating)
    self._start_next_img()

def _on_img_fetch_error(self):

```

```
self._img_active -= 1
self._start_next_img()

# — EPG subtitle loading for channel list —————

def _flush_epg_queue(self, gen: int):
    while self._epg_queue and self._epg_active < 8:
        self._start_next_epg(gen)

def _start_next_epg(self, gen: int):
    if not self._epg_queue:
        return
    row, sid = self._epg_queue.pop(0)
    self._epg_active += 1
    w = Worker(lambda s=sid: self.api.get_short_epg(s, limit=2))
    w.signals.result.connect(
        lambda data, r=row, g=gen: self._on_ch_epg_done(r, data, g))
    w.signals.error.connect(lambda _: self._on_ch_epg_err(gen))
    self.pool.start(w)

def _on_ch_epg_done(self, row: int, data: dict, gen: int):
    self._epg_active -= 1
    if gen == self._load_gen:
        listings = data.get("epg_listings", [])
        if listings:
            cur = listings[0]
            title = _decode_title(cur.get("title", ""))
            item = self.ch_list.item(row)
            if item and title:
                item.setData(EPG_ROLE, title)
                try:
                    s = int(cur.get("start_timestamp") or 0)
                    t = int(cur.get("stop_timestamp") or 0)
                    if s and t:
                        item.setData(EPG_PROG_ROLE, (s, t))
                except Exception:
                    pass
    self._flush_epg_queue(gen)

def _on_ch_epg_err(self, gen: int):
    self._epg_active -= 1
    self._flush_epg_queue(gen)

# — season count badges —————

def _probe_status(self, text: str = ""):
    """Show or hide the probe status bar at the bottom of the poster grid."""
    if text:
        self._probe_status_lbl.setText(text)
        self._probe_status_lbl.setVisible(True)
    elif not self._vod_q_queue and not self._season_queue and self._vod_q_active == 0 and self._season_active == 0:
        self._probe_status_lbl.setVisible(False)

def _flush_season_queue(self, gen: int):
    if self._video_playing:
        return
    while self._season_queue and self._season_active < 3:
        self._start_next_season(gen)

def _start_next_season(self, gen: int):
```

```

if not self._season_queue:
    return
row, sid = self._season_queue.pop(0)
if sid in self._season_cache:
    self._apply_season_badge(row, self._season_cache[sid]["count"], gen)
    cached_q = self._vod_q_cache.get(sid)
    if cached_q:
        # Both count and quality are on disk – nothing to fetch
        self._apply_vod_quality(row, cached_q, gen)
        return # caller's while loop handles the next item
    # Count is cached but quality unknown – fall through to probe
self._season_active += 1
_item = self.poster_grid.item(row)
_name = (_item.data(Qt.ItemDataRole.UserRole) or {}).get("name", "") if _item else ""
remaining = len(self._season_queue)
self._probe_status(f"⦿ Gathering poster info · {_name}" + (f" ({remaining} remaining)" if remaining else ""))
w = Worker(lambda s=sid: self.api.get_series_info(s))
w.signals.result.connect(
    lambda data, r=row, s=sid, g=gen: self._on_season_done(r, s, data, g))
w.signals.error.connect(lambda _, r=row, s=sid, g=gen: self._on_season_err(r, s, g))
self.pool.start(w)

def _on_season_done(self, row: int, sid: str, data: dict, gen: int):
    episodes = data.get("episodes", {})
    count = len(episodes)
    if count:
        import time as _t
        self._season_cache[sid] = {"count": count, "ts": _t.time()}
        self._save_season_cache()

    # extract quality from first episode of first season
    q_label = "N/A"
    eid = ext = None
    try:
        first_season = episodes.get(sorted(episodes.keys())[0], []) if episodes else []
        if first_season:
            ep0 = first_season[0]
            video = ep0.get("info", {}).get("video", {})
            if isinstance(video, list):
                video = video[0] if video else {}
            w = int(video.get("width") or video.get("coded_width", 0) or 0)
            h = int(video.get("height") or video.get("coded_height", 0) or 0)
            if w or h:
                q_label = _stream_quality_label(w, h)
            # Always capture eid/ext so we can probe SD or missing quality
            eid = ep0.get("id")
            ext = ep0.get("container_extension") or "mp4"
    except Exception:
        pass

    self._season_active -= 1
    self._vod_q_cache[sid] = q_label
    self._save_vod_quality_cache()
    if gen == self._load_gen:
        item = self.poster_grid.item(row)
        if item:
            item.setData(VOD_QUALITY_ROLE, q_label)
    self._probe_status()
    self._apply_season_badge(row, count, gen)
    self._flush_season_queue(gen)

```

```

def _on_season_err(self, row: int, sid: str, gen: int):
    self._season_active -= 1
    self._apply_vod_quality(row, "N/A", gen)
    self._probe_status()
    self._flush_season_queue(gen)

def _apply_season_badge(self, row: int, count: int, gen: int):
    if gen != self._load_gen:
        return
    item = self.poster_grid.item(row)
    if item:
        self._refresh_poster_icon(item, gen)

# — VOD quality badges —————

def _flush_vod_q_queue(self, gen: int):
    if self._video_playing:
        return
    while self._vod_q_queue and self._vod_q_active < 3:
        self._start_next_vod_q(gen)

def _start_next_vod_q(self, gen: int):
    if not self._vod_q_queue:
        return
    row, sid = self._vod_q_queue.pop(0)
    cached_q = self._vod_q_cache.get(sid)
    if cached_q:
        # Any cached value (HD, SD, N/A) is the confirmed result — use it
        self._apply_vod_quality(row, cached_q, gen)
        return # caller's while loop handles the next item
    self._vod_q_active += 1
    _item = self.poster_grid.item(row)
    _name = (_item.data(Qt.ItemDataRole.UserRole) or {}).get("name", "") if _item else ""
    remaining = len(self._vod_q_queue)
    self._probe_status(f"⦿ Gathering poster info · {_name}" + (f" ({remaining} remaining)" if remaining else ""))
    w = Worker(lambda s=sid: self.api.get_vod_info(s))
    w.signals.result.connect(
        lambda data, r=row, s=sid, g=gen: self._on_vod_q_done(r, s, data, g))
    w.signals.error.connect(lambda _, r=row, s=sid, g=gen: self._on_vod_q_err(r, s, g))
    self.pool.start(w)

def _on_vod_q_done(self, row: int, sid: str, data: dict, gen: int):
    label = "N/A"
    try:
        video = data.get("info", {}).get("video", {})
        if isinstance(video, list):
            video = video[0] if video else {}
        w = int(video.get("width") or video.get("coded_width", 0) or 0)
        h = int(video.get("height") or video.get("coded_height", 0) or 0)
        if w or h:
            label = _stream_quality_label(w, h)
    except Exception:
        pass
    self._vod_q_active -= 1
    self._vod_q_cache[sid] = label
    self._save_vod_quality_cache()
    self._probe_status()
    self._apply_vod_quality(row, label, gen)
    self._flush_vod_q_queue(gen)

```

```
def _on_vod_q_err(self, row: int, sid: str, gen: int):
    self._vod_q_active -= 1
    self._probe_status()
    self._apply_vod_quality(row, "N/A", gen)
    self._flush_vod_q_queue(gen)

def _apply_vod_quality(self, row: int, label: str, gen: int):
    if gen != self._load_gen:
        return
    item = self.poster_grid.item(row)
    if not item:
        return
    item.setData(VOD_QUALITY_ROLE, label)
    self._refresh_poster_icon(item, gen)

# — live stream quality badges —————

def _flush_live_q_queue(self, gen: int):
    if self._video_playing:
        return
    while self._live_q_queue and self._live_q_active < 6:
        self._start_next_live_q(gen)

def _start_next_live_q(self, gen: int):
    if not self._live_q_queue:
        return
    row, sid, name = self._live_q_queue.pop(0)
    if sid in self._live_q_cache:
        self._apply_live_quality(row, self._live_q_cache[sid], gen)
        self._flush_live_q_queue(gen)
    return

# Read stream data on the main thread NOW before handing off to worker
item = self.ch_list.item(row)
s = (item.data(Qt.ItemDataRole.UserRole) if item else None) or {"name": name}
fallback = _stream_quality_hint(s)

self._live_q_active += 1
url = self.api.live_url(sid)

def _probe(url=url, fallback=fallback):
    import subprocess, json as _json, shutil
    if shutil.which("ffprobe"):
        try:
            result = subprocess.run(
                [
                    "ffprobe", "-v", "quiet",
                    "-print_format", "json",
                    "-show_streams", "-select_streams", "v:0",
                    "-timeout", "8000000", # 8 s in µs
                    url,
                ],
                capture_output=True, text=True, timeout=12,
            )
            data = _json.loads(result.stdout)
            streams = data.get("streams", [])
            if streams:
                w = int(streams[0].get("width", 0) or 0)
                h = int(streams[0].get("height", 0) or 0)
```



```

        if w or h:
            return _stream_quality_label(w, h)
        except Exception:
            pass
        # ffmpeg not available or returned no data - try M3U8 manifest
        import re as _re
        from .xstream_api import _get_session
        try:
            r = _get_session().get(url, timeout=8, allow_redirects=True)
            m = _re.search(r'REOLUTION=(\d+)\x(\d+)', r.text, _re.IGNORECASE)
            if m:
                return _stream_quality_label(int(m.group(1)), int(m.group(2)))
        except Exception:
            pass
        return fallback

    w = Worker(_probe)
    w.signals.result.connect(
        lambda label, r=row, s=sid, g=gen: self._on_live_q_done(r, s, label, g))
    w.signals.error.connect(
        lambda _, r=row, s=sid, fb=fallback, g=gen: self._on_live_q_err(r, s, fb, g))
    self.pool.start(w)

    @staticmethod
    def _load_quality_cache() -> dict:
        try:
            if QUALITY_CACHE_FILE.exists():
                import json as _j
                return _j.loads(QUALITY_CACHE_FILE.read_text())
        except Exception:
            pass
        return {}

    def _save_quality_cache(self):
        try:
            import json as _j
            QUALITY_CACHE_FILE.parent.mkdir(parents=True, exist_ok=True)
            QUALITY_CACHE_FILE.write_text(_j.dumps(self._live_q_cache, indent=2))
        except Exception:
            pass

    @staticmethod
    def _load_vod_quality_cache() -> dict:
        try:
            if VOD_QUALITY_CACHE_FILE.exists():
                import json as _j
                return _j.loads(VOD_QUALITY_CACHE_FILE.read_text())
        except Exception:
            pass
        return {}

    def _save_vod_quality_cache(self):
        try:
            import json as _j
            VOD_QUALITY_CACHE_FILE.parent.mkdir(parents=True, exist_ok=True)
            VOD_QUALITY_CACHE_FILE.write_text(_j.dumps(self._vod_q_cache, indent=2))
        except Exception:
            pass

    @staticmethod

```

```
def _load_season_cache() -> dict:
    try:
        if SEASON_CACHE_FILE.exists():
            import json as _j, time as _t
            cutoff = _t.time() - 90 * 86400
            result = {}
            for sid, v in _j.loads(SEASON_CACHE_FILE.read_text()).items():
                if isinstance(v, dict) and v.get("ts", 0) > cutoff:
                    result[sid] = v
            return result
    except Exception:
        pass
    return {}

def _save_season_cache(self):
    try:
        import json as _j
        SEASON_CACHE_FILE.parent.mkdir(parents=True, exist_ok=True)
        SEASON_CACHE_FILE.write_text(_j.dumps(self._season_cache, indent=2))
    except Exception:
        pass

def _precache_vod_quality(self):
    """After login, silently probe the 30 most-recent uncached movies and series."""
    if not self.api:
        return

    # Don't probe while video is playing (live TV, movie, series) – retry in 60 s
    if self._video_playing:
        QTimer.singleShot(60_000, self._precache_vod_quality)
        return

    api = self.api
    already = set(self._vod_q_cache.keys())

    def _fetch():
        out = {"movies": [], "series": []}
        try:
            movies = api.get_vod_streams()
            uncached = [m for m in movies
                        if m.get("stream_id")
                        and str(m.get("stream_id")) not in already]
            out["movies"] = sorted(
                uncached,
                key=lambda m: int(m.get("added", 0) or 0),
                reverse=True,
            )[:30]
        except Exception:
            pass
        try:
            series_list = api.get_series()
            uncached = [s for s in series_list
                        if s.get("series_id")
                        and str(s.get("series_id")) not in already]
            out["series"] = sorted(
                uncached,
                key=lambda s: int(s.get("last_modified", 0) or 0),
                reverse=True,
            )[:30]
        except Exception:
```

```
        pass
    return out

    w = Worker(_fetch)
    w.signals.result.connect(self._on_precache_vod_fetched)
    w.signals.error.connect(lambda _: None)
    self.pool.start(w)

def _on_precache_vod_fetched(self, data: dict):
    api = self.api
    if not api:
        return

    # Live TV started while the API fetch was in-flight – abort quietly
    if self._playing_stream is not None and self._tab in (0, 3):
        return

    for m in data.get("movies", []):
        sid = str(m.get("stream_id", ""))
        ext = m.get("container_extension") or "mp4"
        url = api.movie_url(sid, ext)

        def _probe_movie(url=url, sid=sid,
                        live=lambda: self._playing_stream is not None and self._tab in (0, 3)):
            if live():
                return sid, None
            import subprocess, json as _j, shutil
            if shutil.which("ffprobe"):
                try:
                    res = subprocess.run(
                        ["ffprobe", "-v", "quiet", "-print_format", "json",
                         "-show_streams", "-select_streams", "v:0",
                         "-timeout", "8000000", url],
                        capture_output=True, text=True, timeout=12,
                    )
                    st = _j.loads(res.stdout).get("streams", [])
                    if st:
                        w = int(st[0].get("width", 0) or 0)
                        h = int(st[0].get("height", 0) or 0)
                        if w or h:
                            return sid, _stream_quality_label(w, h)
                except Exception:
                    pass
            return sid, None

        pw = Worker(_probe_movie)
        pw.signals.result.connect(self._on_precache_vod_result)
        pw.signals.error.connect(lambda _: None)
        self.pool.start(pw)

    for s in data.get("series", []):
        sid = str(s.get("series_id", ""))

        def _probe_series(sid=sid, api=api,
                        live=lambda: self._playing_stream is not None and self._tab in (0, 3)):
            if live():
                return sid, None, 0
            import subprocess, json as _j, shutil
            try:
                info = api.get_series_info(sid)
```

```

        episodes = info.get("episodes", {})
        count = len(episodes)
        if not episodes:
            return sid, None, 0
        first_season = episodes.get(sorted(episodes.keys())[0], [])
        if not first_season:
            return sid, None, count
        ep0 = first_season[0]
        video = ep0.get("info", {}).get("video", {})
        if isinstance(video, list):
            video = video[0] if video else {}
        w = int(video.get("width") or video.get("coded_width", 0) or 0)
        h = int(video.get("height") or video.get("coded_height", 0) or 0)
        if w and h:
            label = _stream_quality_label(w, h)
            if label != "SD":
                return sid, label, count
        eid = ep0.get("id")
        ext = ep0.get("container_extension") or "mp4"
        if not eid:
            return sid, None, count
        url = api.series_episode_url(eid, ext)
        if shutil.which("ffprobe"):
            res = subprocess.run(
                ["ffprobe", "-v", "quiet", "-print_format", "json",
                 "-show_streams", "-select_streams", "v:0",
                 "-timeout", "8000000", url],
                capture_output=True, text=True, timeout=12,
            )
            st = _j.loads(res.stdout).get("streams", [])
            if st:
                pw = int(st[0].get("width", 0) or 0)
                ph = int(st[0].get("height", 0) or 0)
                if pw or ph:
                    return sid, _stream_quality_label(pw, ph), count
        except Exception:
            pass
        return sid, None, 0

    pw = Worker(_probe_series)
    pw.signals.result.connect(self._on_precache_vod_result)
    pw.signals.error.connect(lambda _: None)
    self.pool.start(pw)

def _on_precache_vod_result(self, result):
    if not result:
        return
    sid = result[0]
    label = result[1] if len(result) > 1 else None
    count = result[2] if len(result) > 2 else 0
    if count:
        import time as _t
        self._season_cache[sid] = {"count": count, "ts": _t.time()}
        self._save_season_cache()
    if not label:
        return
    self._vod_q_cache[sid] = label
    self._save_vod_quality_cache()
    self._update_vod_badge(sid, label)

```

```
def _update_vod_badge(self, sid: str, label: str):
    """Update quality badge on poster grid if this movie/series is currently visible."""
    for i in range(self.poster_grid.count()):
        item = self.poster_grid.item(i)
        if not item:
            continue
        s = item.data(Qt.ItemDataRole.UserRole)
        if not isinstance(s, dict):
            continue
        item_sid = str(s.get("stream_id", "") or s.get("series_id", ""))
        if item_sid == sid:
            item.setData(VOD_QUALITY_ROLE, label)
            self._refresh_poster_icon(item, self._load_gen)
            break

def _on_live_q_done(self, row: int, sid: str, label: str, gen: int):
    self._live_q_active -= 1
    if label:
        self._live_q_cache[sid] = label
        self._apply_live_quality(row, label, gen)
        self._save_quality_cache()
    self._flush_live_q_queue(gen)

def _on_live_q_err(self, row: int, sid: str, fallback: str, gen: int):
    self._live_q_active -= 1
    if fallback:
        self._live_q_cache[sid] = fallback
        self._apply_live_quality(row, fallback, gen)
        self._save_quality_cache()
    self._flush_live_q_queue(gen)

def _apply_live_quality(self, row: int, label: str, gen: int):
    if gen != self._load_gen:
        return
    item = self.ch_list.item(row)
    if not item:
        return
    item.setData(QUALITY_ROLE, label)
    self.ch_list.viewport().update()

def _refresh_poster_icon(self, item, gen: int):
    if gen != self._load_gen:
        return
    # Item may have been removed if the category changed – check it's still in the grid
    if self.poster_grid.row(item) < 0:
        return
    base = item.data(BASE_POSTER_ROLE)
    if not base or base.isNull():
        return
    try:
        px = base.copy()
        if px.isNull():
            return
        p = QPainter(px)
        if not p.isActive():
            return
        p.setRenderHint(QPainter.RenderHint.Antialiasing)

        s = item.data(Qt.ItemDataRole.UserRole) or {}
        # season badge – top-right (purple)
```

```

sid = str(s.get("series_id", ""))
count = self._season_cache[sid]["count"] if sid and sid in self._season_cache else 0
if count:
    text = f"{{count}} Season{{'s' if count != 1 else ''}}"
    fb = QFont(); fb.setPixelSize(8); fb.setBold(True)
    p.setFont(fb)
    fm = p.fontMetrics()
    bw = fm.horizontalAdvance(text) + 8
    bh = 14
    bx = _POSTER_W - bw - 4
    by = 4
    p.setBrush(QColor("#7b1fa2")); p.setPen(Qt.PenStyle.NoPen)
    p.drawRoundedRect(bx, by, bw, bh, 3, 3)
    p.setPen(QColor("white"))
    p.drawText(bx, by, bw, bh, Qt.AlignmentFlag.AlignCenter, text)

# fav star - top-left (gold)
if item.data(FAV_ROLE):
    fs = QFont(); fs.setPixelSize(11); fs.setBold(True)
    p.setFont(fs)
    fm2 = p.fontMetrics()
    sw = fm2.horizontalAdvance("★") + 6
    sh = 14
    p.setBrush(QColor("#f9a825")); p.setPen(Qt.PenStyle.NoPen)
    p.drawRoundedRect(4, 4, sw, sh, 3, 3)
    p.setPen(QColor("#111111"))
    p.drawText(4, 4, sw, sh, Qt.AlignmentFlag.AlignCenter, "★")

# API quality badge - bottom-right (colour-coded by quality)
vq = item.data(VOD_QUALITY_ROLE) or ""
if vq:
    fv = QFont(); fv.setPixelSize(9); fv.setBold(True)
    p.setFont(fv)
    fm3 = p.fontMetrics()
    vw = fm3.horizontalAdvance(vq) + 10
    vh = 16
    bx = _POSTER_W - vw - 4
    by = _POSTER_H - vh - 4
    p.setBrush(QColor(_quality_color(vq))); p.setPen(Qt.PenStyle.NoPen)
    p.drawRoundedRect(bx, by, vw, vh, 4, 4)
    p.setPen(QColor("white"))
    p.drawText(bx, by, vw, vh, Qt.AlignmentFlag.AlignCenter, vq)

# watch-progress bar - bottom strip (red)
prog = item.data(WATCH_PROG_ROLE)
if prog:
    wpos = prog.get("pos") or 0
    wdur = prog.get("dur") or 0
    if wdur > 0 and wpos > 0:
        pct = min(1.0, wpos / wdur)
        bar_h = 4
        bar_y = _POSTER_H - bar_h
        p.setPen(Qt.PenStyle.NoPen)
        p.setBrush(QColor("#444444"))
        p.drawRect(0, bar_y, _POSTER_W, bar_h)
        p.setBrush(QColor("#f44336"))
        p.drawRect(0, bar_y, int(_POSTER_W * pct), bar_h)

p.end()
item.setIcon(QIcon(px))

```

```
except Exception:
    pass

def _poster_context_menu(self, pos):
    item = self.poster_grid.itemAt(pos)
    if not item:
        return
    s = item.data(Qt.ItemDataRole.UserRole)
    if not isinstance(s, dict):
        return
    iid = _fav_id(s)
    is_fav = any(_fav_id(x) == iid for x in self.config.favorite_streams)
    menu = QMenu(self)
    menu.setStyleSheet(
        "QMenu{background:#0d1117;color:#d0d8e8;border:1px solid #1e2a3a;}"
        "QMenu::item{padding:8px 20px;}"
        "QMenu::item:selected{background:#1565c0;}"
    )
    act = menu.addAction("★ Remove from Favorites" if is_fav else "☆ Add to Favorites")
    act.triggered.connect(lambda: self._toggle_poster_favorite(s, item, not is_fav))
    menu.exec(self.poster_grid.mapToGlobal(pos))

def _toggle_poster_favorite(self, s: dict, item, add: bool):
    iid = _fav_id(s)
    self.config.favorite_streams = [
        x for x in self.config.favorite_streams if _fav_id(x) != iid
    ]
    if add:
        self.config.favorite_streams.append(s)
        self._save_fav_poster(s, item)
    else:
        self._delete_fav_poster(s)
    self.config.save()

    cur = self.cat_list.currentItem()
    viewing_favs = cur and str(cur.data(Qt.ItemDataRole.UserRole)) == "__FAVORITES__"
    if not add and viewing_favs:
        row = self.poster_grid.row(item)
        if row >= 0:
            self.poster_grid.takeItem(row)
        return

    item.setData(FAV_ROLE, add)
    self.poster_grid.viewport().update()
    self._refresh_poster_icon(item, self._load_gen)

def _save_fav_poster(self, s: dict, item):
    try:
        sid = str(s.get("stream_id", "") or s.get("series_id", ""))
        if not sid:
            return
        base = item.data(BASE_POSTER_ROLE)
        if not base or base.isNull():
            return
        POSTER_CACHE_DIR.mkdir(parents=True, exist_ok=True)
        base.save(str(POSTER_CACHE_DIR / f"{sid}.jpg"), "JPEG", 85)
    except Exception:
        pass

def _delete_fav_poster(self, s: dict):
```

```

try:
    sid = str(s.get("stream_id", "") or s.get("series_id", ""))
    if sid:
        p = POSTER_CACHE_DIR / f"{sid}.jpg"
        if p.exists():
            p.unlink()
except Exception:
    pass

def eventFilter(self, obj, event):
    from PyQt6.QtCore import QEvent
    from .delegates import ChannelDelegate as _CD
    if event.type() == QEvent.Type.MouseButtonPress:
        pos = event.pos()

        # — poster grid star click —————
        if obj is self.poster_grid.viewport():
            item = self.poster_grid.itemAt(pos)
            if item:
                idx = self.poster_grid.indexFromItem(item)
                cell = self.poster_grid.visualRect(idx)
                if self._poster_delegate.star_rect(cell).contains(pos):
                    s = item.data(Qt.ItemDataRole.UserRole)
                    if isinstance(s, dict):
                        self._toggle_poster_favorite(s, item, not bool(item.data(FAV_ROLE)))
                    return True

        # — channel list star click —————
        elif obj is self.ch_list.viewport():
            item = self.ch_list.itemAt(pos)
            if item:
                s = item.data(Qt.ItemDataRole.UserRole)
                if isinstance(s, dict) and not s.get("__hidden_cat__"):
                    idx = self.ch_list.indexFromItem(item)
                    cell = self.ch_list.visualRect(idx)
                    # Reconstruct the icon rect the delegate uses
                    from PyQt6.QtCore import QRect as _R
                    ix = cell.left() + 8
                    iy = cell.top() + (cell.height() - _LOGO_SZ) // 2 - 4
                    icon_rect = _R(ix, iy, _LOGO_SZ, _LOGO_SZ)
                    if _CD.star_rect_for(icon_rect).contains(pos):
                        sid = s.get("stream_id")
                        if self._tab == 3:
                            is_fav = any(x.get("stream_id") == sid for x in self.config.radio_favorite_streams)
                        else:
                            is_fav = any(x.get("stream_id") == sid for x in self.config.favorite_streams)
                        self._toggle_favorite(s, not is_fav)
                        item.setData(FAV_ROLE, not is_fav)
                        self.ch_list.viewport().update()
                    return True

    return super().eventFilter(obj, event)

def _refresh_epg(self):
    """Force-reload EPG for every channel currently visible in the Live TV list."""
    if not self.api or self._tab not in (0, 3):
        return
    count = self.ch_list.count()
    if count == 0:
        return

```



```

# Clear existing EPG subtitles and progress bars
for i in range(count):
    item = self.ch_list.item(i)
    if item:
        item.setData(EPG_ROLE, "")
        item.setData(EPG_PROG_ROLE, None)
self.ch_list.update()
# Cancel any pending (not yet started) EPG fetches and re-queue all channels
self._epg_queue.clear()
gen = self._load_gen
for i in range(count):
    item = self.ch_list.item(i)
    if item:
        s = item.data(Qt.ItemDataRole.UserRole)
        if isinstance(s, dict) and s.get("stream_id"):
            self._epg_queue.append((i, s.get("stream_id")))
self._flush_epg_queue(gen)
self._status(f"Refreshing EPG for {len(self._epg_queue)} channels...")

def _on_img(self, row, pixmap, gen, key, tw, th, name, rating):
    if pixmap.isNull():
        return
    scaled = pixmap.scaled(tw, th,
                           Qt.AspectRatioMode.KeepAspectRatioByExpanding,
                           Qt.TransformationMode.SmoothTransformation)
    x = (scaled.width() - tw) // 2
    y = (scaled.height() - th) // 2
    scaled = scaled.copy(x, y, tw, th)
    if tw == _POSTER_W:
        scaled = self._render_overlay(scaled, name, rating)
    self._img_cache[key] = scaled
    self._apply_img(row, scaled, gen)

def _apply_img(self, row, pixmap, gen):
    if gen != self._load_gen:
        return
    lst = self.ch_list if self._tab in (0, 3) else self.poster_grid
    item = lst.item(row)
    if item:
        if self._tab in (1, 2):
            item.setData(BASE_POSTER_ROLE, pixmap)
            self._refresh_poster_icon(item, gen)
        else:
            item.setIcon(QIcon(pixmap))

def _render_overlay(self, px: QPixmap, name: str, rating: str) -> QPixmap:
    result = px.copy()
    if result.isNull():
        return px
    p = QPainter(result)
    if not p.isActive():
        return px
    p.setRenderHint(QPainter.RenderHint.Antialiasing)
    w, h = result.width(), result.height()

    # dark gradient at bottom
    grad = QLinearGradient(0, h - 55, 0, h)
    grad.setColorAt(0, QColor(0, 0, 0, 0))
    grad.setColorAt(1, QColor(0, 0, 0, 220))
    p.fillRect(0, h - 55, w, 55, grad)

```

```

# title text (up to 2 lines)
p.setPen(QColor("#ffffff"))
fn = QFont(); fn.setPixelSize(11); fn.setBold(True)
p.setFont(fn)
fm = p.fontMetrics()
l1, l2 = _wrap_text(name, fm, w - 8)
if l2:
    p.drawText(4, h - 29, w - 8, 14, Qt.AlignmentFlag.AlignLeft, l1)
    p.drawText(4, h - 13, w - 8, 14, Qt.AlignmentFlag.AlignLeft, l2)
else:
    p.drawText(4, h - 17, w - 8, 14, Qt.AlignmentFlag.AlignLeft, l1)

# quality badge – bottom-left corner
quality = _quality_from_name(name)
if quality:
    fq = QFont(); fq.setPixelSize(9); fq.setBold(True)
    p.setFont(fq)
    fm_q = p.fontMetrics()
    qw = fm_q.horizontalAdvance(quality) + 8
    qh = 14
    qx, qy = 4, h - qh - 4
    p.setBrush(QColor("#1565c0")); p.setPen(Qt.PenStyle.NoPen)
    p.drawRoundedRect(qx, qy, qw, qh, 3, 3)
    p.setPen(QColor("white"))
    p.drawText(qx, qy, qw, qh, Qt.AlignmentFlag.AlignCenter, quality)

p.end()
return result

# — playback —————

def _on_channel_activated(self, item: QListWidgetItem):
    if self._ch_select_mode:
        s = item.data(Qt.ItemDataRole.UserRole)
        if s and not s.get("__hidden_cat__") and s.get("stream_id"):
            new_state = (Qt.CheckState.Unchecked
                          if item.checkState() == Qt.CheckState.Checked
                          else Qt.CheckState.Checked)
            item.setCheckState(new_state)
            self.ch_list.viewport().update()
            self._update_ch_select_bar()
        return
    s = item.data(Qt.ItemDataRole.UserRole)
    if not s or s.get("__hidden_cat__"):
        return
    sid = s.get("stream_id")

    if self._tab in (0, 3):
        if self._tab == 3:
            is_fav = any(x.get("stream_id") == sid for x in self.config.radio_favorite_streams)
            quality = ""
        else:
            is_fav = any(x.get("stream_id") == sid for x in self.config.favorite_streams)
            quality = item.data(QUALITY_ROLE) or ""
        if self._tab == 3:
            # Radio – center the logo, no synopsis
            popup = _VodPopup(s, item, is_fav, quality, parent=self, show_plot=False)
        else:

```

```

        epg_title = item.data(EPG_ROLE) or ""
        epg_desc = ""
        if epg_title and str(sid) == str((self._playing_stream or {}).get("stream_id", "")):
            epg_desc = self.epg_desc.text() if self.epg_desc.isVisible() else ""
        epg_info = (epg_title + ("\n" + epg_desc if epg_desc else "")) if epg_title else ""
        popup = _VodPopup(s, item, is_fav, quality, parent=self, synopsis=epg_info)
        popup.exec()
        if popup.fav_toggled:
            self._toggle_favorite(s, not is_fav)
            cur_cat = self.cat_list.currentItem()
            viewing_favs = cur_cat and str(cur_cat.data(Qt.ItemDataRole.UserRole)) == "__FAVORITES__"
            if is_fav and viewing_favs:
                row = self.ch_list.row(item)
                if row >= 0:
                    self.ch_list.takeItem(row)
            else:
                item.setData(FAV_ROLE, not is_fav)
                self.ch_list.viewport().update()
        if not popup.play_clicked:
            return

    url = self.api.live_url(sid)
    self._play(s, url)
    # prepend to recent, deduplicate, cap at 30
    self.config.recent_streams = [
        x for x in self.config.recent_streams if x.get("stream_id") != sid
    ]
    self.config.recent_streams.insert(0, s)
    self.config.recent_streams = self.config.recent_streams[:30]
    self.config.save()
    # fetch EPG
    w = Worker(lambda: self.api.get_short_epg(sid))
    w.signals.result.connect(self._on_epg)
    w.signals.error.connect(lambda _: None)
    self.pool.start(w)

def _on_vod_activated(self, item: QListWidgetItem):
    s = item.data(Qt.ItemDataRole.UserRole)
    if not s:
        return

    if self._tab == 2:
        watch_key = s.get("_watch_key", "")
        is_fav = any(_fav_id(x) == _fav_id(s) for x in self.config.favorite_streams)
        quality = item.data(VOD_QUALITY_ROLE) or ""
        popup = _VodPopup(s, item, is_fav, quality, parent=self)
        popup.exec()
        if popup.fav_toggled:
            self._toggle_poster_favorite(s, item, not is_fav)
        if not popup.play_clicked:
            return
        if watch_key.startswith("ep:"):
            eid = s.get("_ep_id") or watch_key[3:]
            ext = s.get("_ep_ext", "mp4")
            url = self.api.series_episode_url(eid, ext)
            self._middle_stack.hide()
            self._splitter.widget(2).show()
            self._splitter.setSizes([210, 0, 1000])
            self._vod_title_lbl.setText(s.get("name", ""))
            self._back_bar.show()

```

```

        QTimer.singleShot(0, lambda: self._play(s, url, vod_key=watch_key))
        return
    self._open_series(s)
    return

    sid = s.get("stream_id")
    ext = s.get("container_extension", "mp4")
    key = f"vod:{sid}"

    is_fav = any(_fav_id(x) == _fav_id(s) for x in self.config.favorite_streams)
    quality = item.data(VOD_QUALITY_ROLE) or ""
    popup = _VodPopup(s, item, is_fav, quality, parent=self, show_plot=False)
    popup.exec()

    # Handle favorites toggle (whether or not Play was pressed)
    if popup.fav_toggled:
        self._toggle_poster_favorite(s, item, not is_fav)

    if not popup.play_clicked:
        return

    url = self.api.movie_url(sid, ext)
    self._middle_stack.hide()
    self._splitter.widget(2).show()
    self._splitter.setSizes([210, 0, 1000])
    self._vod_title_lbl.setText(s.get("name", ""))
    self._back_bar.show()
    QTimer.singleShot(0, lambda: self._play(s, url, vod_key=key))

def _open_series(self, s: dict):
    sid = s.get("series_id") or s.get("stream_id")
    name = s.get("name", "Series")
    self._status(f>Loading {name}...")
    w = Worker(lambda: self.api.get_series_info(sid))
    w.signals.result.connect(lambda data: self._show_series_dialog(s, data))
    w.signals.error.connect(lambda e: self._status(f>Error loading series: {e}"))
    self.pool.start(w)

def _show_series_dialog(self, s: dict, data: dict):
    episodes_by_season = data.get("episodes", {})
    if not episodes_by_season:
        self._status("No episodes found for this series")
        return
    cover_url = s.get("cover") or s.get("stream_icon") or ""
    series_info = data.get("info", {})
    dlg = SeriesDialog(
        s.get("name", "Series"), episodes_by_season, self.api,
        series_info=series_info, cover_url=cover_url, parent=self,
    )
    dlg.setStyleSheet(self.styleSheet())
    if dlg.exec() and dlg.selected_url:
        ep_stream = {
            "name": dlg.selected_name or s.get("name", ""),
            "_ep_id": getattr(dlg, "_selected_ep_id", ""),
            "_ep_ext": getattr(dlg, "_selected_ep_ext", "mp4"),
            "stream_icon": s.get("cover") or s.get("stream_icon") or "",
            "_series_name": s.get("name", ""),
            "series_id": s.get("series_id", ""),
        }
    self._middle_stack.hide()

```

```

        self._splitter.widget(2).show()
        self._splitter.setSizes([210, 0, 1000])
        self._vod_title_lbl.setText(ep_stream["name"])
        self._back_bar.show()
        url = dlg.selected_url
        eid = getattr(dlg, "_selected_ep_id", "")
        key = f"ep:{eid}"
        self._series_state = {
            "data": episodes_by_season,
            "seasons": dlg.seasons_list,
            "season_idx": dlg.selected_season_idx,
            "ep_row": dlg.selected_ep_row,
            "series_name": s.get("name", ""),
        }
        self._next_ep_btn.show()
        QTimer.singleShot(0, lambda: self._play(ep_stream, url, vod_key=key))

def _play(self, s: dict, url: str, vod_key: str = "", content_type: str = ""):
    self._progress_timer.stop()
    self._current_vod_key = vod_key or None

    self._playing_stream = s
    self._last_url = url
    _is_radio = self._is_radio(s)
    _display_name = (" " if _is_radio else "") + self._dn(s.get("name", ""))
    self.now_playing.setText(_display_name)
    self.play_btn.setText("")
    self._epg_bar.hide()
    log = AppLogger.get()
    log.info(f"Playing: {s.get('name','')}\n          URL: {url}")

    # Determine content type if not supplied
    if not content_type:
        if vod_key:
            content_type = "series" if vod_key.startswith("ep:") else "movies"
        else:
            content_type = "live"

    # Check player preference
    from .player_select_screen import (launch_external, VlcRcMonitor,
                                       VlcTcpMonitor, _find_free_port)
    player_id = self.config.player_prefs.get(content_type, "builtin")
    if player_id and player_id != "builtin":
        # Kill previous external player instance if still running
        self._vlc_click_timer.stop()
        if self._vlc_monitor is not None:
            self._vlc_monitor.stop()
            self._vlc_monitor = None
        if self._ext_proc is not None:
            try:
                if self._ext_proc.poll() is None:
                    self._ext_proc.terminate()
            except Exception:
                pass
            self._ext_proc = None

    pname = player_id.replace("\\", "/").split("/")[1].lower()
    if "vlc" in pname:
        # Embed VLC into the player panel
        self._player_stack.setCurrentIndex(1)

```

```

        self.player.terminate_mpv() # free the window handle for VLC
        wid = int(self.player.winId())
        self._vlc_embedded = True
    else:
        wid = 0
        self._vlc_embedded = False

    vlc_rc_port = (_find_free_port()
                    if self._vlc_embedded and sys.platform == "win32" else 0)
    self._ext_proc = launch_external(
        player_id, url, self.config.extra_players,
        width=self.player.width(), height=self.player.height(), wid=wid,
        vlc_rc_port=vlc_rc_port)
    log.info(f"Launched external player '{player_id}' for {s.get('name','')}")
    # External players emit no MPV signals, so auto-hide after a delay.
    # Radio is audio-only – keep the panel black (no overlay).
    if self._vlc_embedded and self._tab != 3:
        self.player.show_message(
            "Some streams take longer than others.\nPlease be patient.",
            auto_hide_ms=2000,
        )
    badge_name = os.path.splitext(os.path.basename(player_id.replace("\\", "/")))[0].upper()
    self._player_badge.setText(f"▶ {badge_name}")
    self._player_badge.setStyleSheet(
        "color:#ffa726; font-size:10px; font-weight:bold;"
        " background:transparent; border:none; padding:2px 4px;"
    )
    self._player_badge.show()
    # Disable controls that only work with the built-in player
    self._cc_btn.setEnabled(False)
    self._cc_btn.setToolTip("Subtitles not available for external players")
    self._seek_bar.hide()
    self._pos_timer.stop()
    # Drive the stream-health dot for embedded VLC on both platforms.
    # Windows: RC over TCP; Linux: RC over stdin/stdout.
    if self._vlc_embedded and self._tab != 3 and self._ext_proc is not None:
        if sys.platform == "win32":
            self._vlc_monitor = VlcTcpMonitor(vlc_rc_port)
        else:
            self._vlc_monitor = VlcRcMonitor(self._ext_proc)
        self._buf_last_pos = 0.0
        self._buf_started = False
        self._buf_stall_ticks = 0
        self._buf_stream_ticks = 0
        self._failure_recorded = False
        self._success_cleared = False
        self._set_buf_state("connecting")
        self._buf_timer.start()
    else:
        self._buf_timer.stop()

    # Start click polling for embedded VLC on Windows.
    if self._vlc_embedded and sys.platform == "win32":
        self._vlc_btn_was_down = False
        self._vlc_last_click = 0.0
        self._vlc_click_timer.start()
    else:
        self._vlc_click_timer.stop()
        self._buf_indicator.hide()
    return

```

```

# Re-enable built-in-only controls
self._cc_btn.setEnabled(True)
self._player_badge.setText("► BUILT-IN")
self._player_badge.setStyleSheet(
    "color:#42a5f5; font-size:10px; font-weight:bold;"
    " background:transparent; border:none; padding:2px 4px;"
)
self._player_badge.show()
self._player_stack.setCurrentIndex(1)
# Clear any previous frame to black before loading the new stream, so
# switching channels never flashes the old video. This also resets
# time_pos so the patience overlay isn't dismissed by a stale value.
self.player.stop()
# Radio is audio-only – keep the panel black at all times (no overlay).
if self._tab == 3:
    self.player.hide_message()
else:
    self.player.show_message(
        "Some streams take longer than others.\nPlease be patient."
    )
self.player.play(url, buffer_secs=self.config.buffer_secs,
                 is_live=self._tab in (0, 3))
# New stream: reset progress tracking and start at CONNECTING (not green
# – green is only earned once playback actually advances).
self._buf_stall_ticks = 0
self._buf_last_pos = 0.0
self._buf_started = False
self._buf_stream_ticks = 0
self._failure_recorded = False
self._success_cleared = False
self._set_buf_state("connecting")
self._buf_timer.start()
if self.player.error:
    log.error(f"MPV error: {self.player.error}")
elif self.player._mpv is None:
    log.error("MPV not initialized – player panel may not be visible yet")
self._status(f"Playing: {s.get('name','')}")

# dim CC button immediately; refresh once MPV has detected tracks (~2s)
self._cc_btn.setStyleSheet(
    "QPushButton#icon_btn{color:#2a3a4a;background:transparent;border:none;"
    "border-radius:4px;font-size:11px;font-weight:bold;padding:4px 8px;}"
    "QPushButton#icon_btn:hover{color:#4a6a8a;background:#111827;}"
)
self._cc_btn.setToolTip("Detecting subtitle tracks...")
QTimer.singleShot(2500, self._refresh_cc_btn)

# seek bar: show for VOD/episodes (has duration), hide for live TV
if vod_key:
    self._seek_bar.show()
    self._seek_slider.setValue(0)
    self._pos_lbl.setText("0:00")
    self._dur_lbl.setText("0:00")
    self._pos_timer.start()
else:
    self._seek_bar.hide()
    self._pos_timer.stop()

# apply saved CC preference after MPV has loaded tracks

```

```

if self.config.cc_enabled:
    QTimer.singleShot(2000, self._apply_saved_cc)
else:
    self.player.set_sub_track(0)
    self._set_cc_style(False)

if vod_key:
    saved = self.config.watch_progress.get(vod_key, {})
    pos = saved.get("pos", 0)
    if pos > 10:
        def _ask_resume(p=pos, k=vod_key):
            mins, secs = divmod(int(p), 60)
            hrs, mins = divmod(mins, 60)
            ts = f"{hrs}:{mins:02d}:{secs:02d}" if hrs else f"{mins}:{secs:02d}"
            ans = QMessageBox.question(
                self, "Continue Watching",
                f"Resume from {ts}?",
                QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No,
            )
            if ans == QMessageBox.StandardButton.Yes:
                self.player.seek_to(p)
        QTimer.singleShot(1500, _ask_resume)
    self._progress_timer.start()

def _on_epg(self, data: dict):
    listings = data.get("epg_listings", [])
    if not listings:
        return
    now = listings[0]
    nxt = listings[1] if len(listings) > 1 else None
    title = _decode_title(now.get("title", ""))
    s = _fmt_ts(now.get("start_timestamp"))
    e = _fmt_ts(now.get("stop_timestamp"))
    self.epg_now.setText(f"NOW: {title} {s} - {e}")
    if nxt:
        nt = _decode_title(nxt.get("title", ""))
        ns = _fmt_ts(nxt.get("start_timestamp"))
        self.epg_next.setText(f"NEXT: {nt} {ns}")
    else:
        self.epg_next.setText("")
    desc = _decode_title(now.get("description", ""))
    if desc:
        self.epg_desc.setText(desc)
        self.epg_desc.show()
    else:
        self.epg_desc.hide()
    self._epg_bar.show()

# Sync the channel list item's EPG subtitle with the fresh data
if self._playing_stream and title:
    playing_sid = str(self._playing_stream.get("stream_id", ""))
    for i in range(self.ch_list.count()):
        item = self.ch_list.item(i)
        if item is None:
            continue
        s_data = item.data(Qt.ItemDataRole.UserRole)
        if isinstance(s_data, dict) and str(s_data.get("stream_id", "")) == playing_sid:
            item.setData(EPG_ROLE, title)
            self.ch_list.viewport().update()
            break

```



```

def _filter_content(self, text: str):
    lst = self.ch_list if self._tab in (0, 3) else self.poster_grid
    for i in range(lst.count()):
        item = lst.item(i)
        s = item.data(Qt.ItemDataRole.UserRole)
        name = s.get("name", "") if isinstance(s, dict) else item.text()
        item.setHidden(bool(text) and text.lower() not in name.lower())

def _toggle_fullscreen(self):
    if self._playing_stream:
        self._toggle_hub_fs()

def _toggle_hub_fs(self):
    """Toggle fullscreen for the player panel – works for every player."""
    import time
    from .app_logger import AppLogger
    log = AppLogger.get()
    now = time.monotonic()
    if self._hub_fs:
        # Exiting fullscreen: block if we entered less than 0.8 s ago.
        # showFullScreen() returns before the window manager applies the
        # state change, so deferred X11/Qt synthetic events can re-trigger
        # this exit path almost immediately. The 0.8 s guard absorbs them
        # while still feeling instant to the user.
        enter_t = getattr(self, '_hub_fs_enter_t', 0.0)
        elapsed = now - enter_t
        log.info(f"[FS] exit requested, elapsed={elapsed:.3f}s")
        if elapsed < 0.8:
            log.info("[FS] exit suppressed (too soon after enter)")
            return
    else:
        self._hub_fs_enter_t = now
        log.info("[FS] entering fullscreen")
        self._toggle_hub_fs_inner()

def _toggle_hub_fs_inner(self):
    if not self._hub_fs:
        self._hub_fs = True
        self._pre_hub_fs_state = self.windowState()
        self._pre_hub_fs_sizes = self._splitter.sizes()
        self._pre_hub_fs_backbar = self._back_bar.isVisible()
        self._pre_hub_fs_seekbar = self._seek_bar.isVisible()
        self._pre_hub_fs_epgbar = self._epg_bar.isVisible()
        # Hide all UI chrome – only the player panel remains
        self._content_topbar.hide()
        self._back_bar.hide()
        self._epg_bar.hide()
        self._seek_bar.hide()
        self._controls_bar.hide()
        self._splitter.widget(0).hide()
        self._splitter.widget(1).hide()
        self.showFullScreen()
    else:
        self._hub_fs = False
        self._content_topbar.show()
        self._controls_bar.show()
        if self._pre_hub_fs_backbar:
            self._back_bar.show()

```

```
        if self._pre_hub_fs_seekbar:
            self._seek_bar.show()
        if self._pre_hub_fs_epgbar:
            self._epg_bar.show()
        self._splitter.widget(0).show()
        self._splitter.widget(1).show()
        state = self._pre_hub_fs_state
        self._pre_hub_fs_state = None
        if state is not None:
            self.setWindowState(state)
        else:
            self.showNormal()
        self._splitter.setSizes(self._pre_hub_fs_sizes)

def _toggle_pause(self):
    if not self._playing_stream:
        return
    if self._ext_proc is not None:
        if self._vlc_embedded:
            try:
                if self._ext_proc.poll() is None and self._ext_proc.stdin:
                    self._ext_proc.stdin.write(b"pause\n")
                    self._ext_proc.stdin.flush()
                    self._vlc_paused = not self._vlc_paused
                    self.play_btn.setText("▶" if self._vlc_paused else "")
            except Exception:
                pass
        else:
            self._status("Pause not available for this external player")
            return
    self.player.toggle_pause()
    self.play_btn.setText("▶" if self.player.paused else "")

def _update_seek_bar(self):
    if self._seek_dragging:
        return
    dur = self.player.duration
    pos = self.player.time_pos
    if dur > 0:
        self._seek_slider.setValue(int(pos / dur * 1000))
        self._pos_lbl.setText(_fmt_secs(pos))
        self._dur_lbl.setText(_fmt_secs(dur))

def _on_seek_released(self):
    self._seek_dragging = False
    dur = self.player.duration
    if dur > 0:
        pos = self._seek_slider.value() / 1000.0 * dur
        self.player.seek_to(pos)

def _stop(self):
    item = self.ch_list.currentItem()
    if item:
        name = item.data(Qt.ItemDataRole.DisplayRole) or ""
        item.setData(QUALITY_ROLE, _quality_from_name(name))
    self._save_progress()
    self._progress_timer.stop()
    self._pos_timer.stop()
    self._buf_timer.stop()
    self._buf_indicator.hide()
```

```

self._buf_stall_ticks = 0
self.player.hide_message()
self._seek_bar.hide()
self._current_vod_key = None
if self._vlc_monitor is not None:
    self._vlc_monitor.stop()
    self._vlc_monitor = None
if self._ext_proc is not None:
    try:
        if self._ext_proc.poll() is None:
            self._ext_proc.terminate()
    except Exception:
        pass
    self._ext_proc = None
if self._hub_fs:
    self._toggle_hub_fs()
self._vlc_embedded = False
self._vlc_paused = False
self.player.stop()
self._player_stack.setCurrentIndex(0)
self._playing_stream = None
# Resume any badge probing that was paused while video was playing.
gen = self._load_gen
self._flush_season_queue(gen)
self._flush_vod_q_queue(gen)
self._flush_live_q_queue(gen)
self.now_playing.setText("No channel selected")
self._player_badge.hide()
self.play_btn.setText("")
self._epg_bar.hide()
self._cc_btn.setEnabled(True)
self._cc_btn.setStyleSheet("")
self._cc_btn.setToolTip("Toggle closed captions")

# — sleep timer —————

def _toggle_sleep_timer(self):
    if self._sleep_timer.isActive():
        self._sleep_timer.stop()
        self._sleep_remaining = 0
        self._sleep_btn.setText("")
        self._sleep_btn.setToolTip("Sleep timer")
        return
    menu = QMenu(self)
    menu.setStyleSheet(
        "QMenu{background:#0d1117;color:#d0d8e8;border:1px solid #1e2a3a;}"
        "QMenu::item{padding:8px 20px;}"
        "QMenu::item:selected{background:#1565c0;}"
    )
    for mins in [15, 30, 45, 60, 90]:
        menu.addAction(f"{mins} minutes").setData(mins)
    chosen = menu.exec(self._sleep_btn.mapToGlobal(
        self._sleep_btn.rect().bottomLeft()))
    if chosen and chosen.data():
        self._sleep_remaining = int(chosen.data()) * 60
        self._sleep_timer.start()
        self._sleep_btn.setToolTip("Sleep timer — click to cancel")

def _sleep_tick(self):
    self._sleep_remaining -= 1

```

```
        if self._sleep_remaining <= 0:
            self._sleep_timer.stop()
            self._sleep_btn.setText("")
            self._stop()
        else:
            mins, secs = divmod(self._sleep_remaining, 60)
            self._sleep_btn.setText(f"{mins}:{secs:02d}")

# — audio / subtitle tracks —————

def _show_tracks_menu(self):
    menu = QMenu(self)
    menu.setStyleSheet(
        "QMenu{background:#0d1117;color:#d0d8e8;border:1px solid #1e2a3a;}"
        "QMenu::item{padding:8px 20px;}"
        "QMenu::item:selected{background:#1565c0;}"
    )
    audio = self.player.audio_tracks()
    if audio:
        menu.addSection("Audio Tracks")
        for t in audio:
            tid = t.get("id", 0)
            lang = t.get("lang") or ""
            title = t.get("title") or ""
            label = f" {tid}. {title or lang or 'Track ' + str(tid)}"
            act = menu.addAction(label)
            act.setData(("audio", tid))
    subs = self.player.sub_tracks()
    if subs:
        menu.addSection("Subtitles")
        none_act = menu.addAction(" Off")
        none_act.setData(("sub", 0))
        for t in subs:
            tid = t.get("id", 0)
            lang = t.get("lang") or ""
            title = t.get("title") or ""
            label = f" {tid}. {title or lang or 'Sub ' + str(tid)}"
            act = menu.addAction(label)
            act.setData(("sub", tid))
    if not audio and not subs:
        menu.addAction("No tracks available").setEnabled(False)

from PyQt6.QtWidgets import QApplication
cursor_pos = QApplication.instance().overrideCursor()
chosen = menu.exec(self._sleep_btn.mapToGlobal(
    self._sleep_btn.rect().topLeft()))
if chosen and chosen.data():
    kind, tid = chosen.data()
    if kind == "audio":
        self.player.set_audio_track(tid)
    else:
        self.player.set_sub_track(tid)

# — closed captions —————

def _toggle_cc(self):
    subs = self.player.sub_tracks()
    if not subs:
        self._status("No subtitle tracks available for this stream")
    return
```

```

        if self.player.sub_enabled:
            self.player.set_sub_track(0)
            self.config.cc_enabled = False
            self._status("Subtitles off")
        else:
            self.player.set_sub_track(subs[0].get("id", 1))
            self.config.cc_enabled = True
            self._status(f"Subtitles: {subs[0].get('lang') or subs[0].get('title') or 'Track 1'}")
        self.config.save()
        self._refresh_cc_btn()

    def _apply_saved_cc(self):
        subs = self.player.sub_tracks()
        if subs:
            self.player.set_sub_track(subs[0].get("id", 1))
            self._refresh_cc_btn()

    def _refresh_cc_btn(self):
        """Update CC button to reflect track availability and on/off state."""
        subs = self.player.sub_tracks()
        if not subs:
            # no subtitle tracks detected
            self._cc_btn.setStyleSheet(
                "QPushButton#icon_btn{color:#2a3a4a;background:transparent;border:none;"
                "border-radius:4px;font-size:11px;font-weight:bold;padding:4px 8px;}"
                "QPushButton#icon_btn:hover{color:#4a6a8a;background:#111827;}"
            )
            self._cc_btn.setToolTip("No subtitles available for this stream")
        elif self.player.sub_enabled:
            self._cc_btn.setStyleSheet(
                "QPushButton#icon_btn{background:#1565c0;color:white;border:none;"
                "border-radius:4px;font-size:11px;font-weight:bold;padding:4px 8px;}"
                "QPushButton#icon_btn:hover{background:#1976d2;}"
            )
            self._cc_btn.setToolTip("Subtitles ON – click to disable")
        else:
            self._cc_btn.setStyleSheet("") # revert to global stylesheet
            self._cc_btn.setToolTip("Subtitles available – click to enable")

    def _set_cc_style(self, active: bool):
        if active:
            self._cc_btn.setStyleSheet(
                "QPushButton#icon_btn{background:#1565c0;color:white;border:none;"
                "border-radius:4px;font-size:11px;font-weight:bold;padding:4px 8px;}"
                "QPushButton#icon_btn:hover{background:#1976d2;}"
            )
        else:
            self._cc_btn.setStyleSheet("") # revert to global stylesheet

# — next episode —————

    def _go_next_episode(self):
        if not self._series_state:
            return

        st = self._series_state
        seasons = st["seasons"]
        season_idx = st["season_idx"]
        ep_row = st["ep_row"] + 1

        season_key = seasons[season_idx]

```

```

eps = sorted(st["data"].get(season_key, []),
              key=lambda e: e.get("episode_num", 0))

if ep_row >= len(eps):
    # advance to the next season
    season_idx += 1
    if season_idx >= len(seasons):
        self._status("No more episodes in this series")
        self._next_ep_btn.hide()
        self._series_state = None
        return
    season_key = seasons[season_idx]
    eps = sorted(st["data"].get(season_key, []),
                  key=lambda e: e.get("episode_num", 0))
    ep_row = 0

ep = eps[ep_row]
eid = ep.get("id")
ext = ep.get("container_extension", "mp4")
url = self.api.series_episode_url(eid, ext)
num = ep.get("episode_num", "?")
title = ep.get("title") or f"Episode {num}"
prefix = f"E{int(num):02d}" if str(num).isdigit() else f"E{num}"
name = f"{st['series_name']} {prefix} {title}"

st["season_idx"] = season_idx
st["ep_row"] = ep_row

self._vod_title_lbl.setText(name)
self._play({"name": name}, url, vod_key=f"ep:{eid}")

# — stream info —————

def _show_stream_info(self):
    if not self._playing_stream:
        self._status("No stream info available")
        return
    name = self._playing_stream.get("name", "-")
    if self._ext_proc is not None:
        player_label = self._player_badge.text().replace("> ", "").strip()
        url = getattr(self, "_last_url", "")
        short_url = (url[:80] + "...") if len(url) > 80 else url
        self._status(f"Player: {player_label} | Stream: {name} | {short_url}")
    else:
        info = self.player.stream_info()
        if info:
            parts = " | ".join(f"{k}: {v}" for k, v in info.items())
            self._status(f"Stream: {name} | {parts}")
        else:
            self._status(f"Stream: {name} | Built-in | (codec info not yet available)")

# — mini player —————

def _open_mini_player(self):
    if not self._playing_stream or not hasattr(self, '_last_url'):
        self._status("Nothing is playing")
        return
    if self._ext_proc is not None:
        self._status("Mini-player is not available when using an external player")
        return

```

```
    if self._mini_player:
        self._mini_player.close()
    name = self._playing_stream.get("name", "")
    self.player.stop()
    self._mini_player = _MiniPlayer(self._last_url, name, parent=None)
    self._mini_player.closed.connect(self._on_mini_closed)
    self._mini_player.show()

def _on_mini_closed(self):
    self._mini_player = None
    if self._playing_stream and hasattr(self, '_last_url'):
        self.player.play(self._last_url)

# — buffer status indicator —————

def _poll_buffer(self):
    if not self._playing_stream:
        return
    if self._ext_proc is not None:
        if self._vlc_monitor is not None:
            self._poll_vlc()
        return

    # If the user paused on purpose, that is not a stall – leave it green.
    if self.player.paused:
        self._buf_stall_ticks = 0
        return

    # Base the indicator on real playback PROGRESS, not merely on whether
    # MPV is currently waiting for its cache. A failed stream (e.g. HTTP
    # 403) never advances time_pos, so it must not read as a live stream.
    pos = self.player.time_pos
    progressing = pos > (self._buf_last_pos + 0.01)
    self._buf_last_pos = pos

    if progressing:
        self._buf_started = True
        self._buf_stall_ticks = 0
        self._buf_stream_ticks += 1
        self.player.hide_message() # genuine signal – stream is flowing
        self._set_buf_state("stream")
        # Clear failure history after 30 consecutive seconds of good streaming.
        if self._buf_stream_ticks == 30 and not self._success_cleared:
            sid = str((self._playing_stream or {}).get("stream_id", ""))
            if sid and self.config.is_stream_unreliable(sid):
                self._success_cleared = True
                self.config.clear_stream_failure(sid)
                self._update_channel_fail_icon(sid, False)
        return

    # No progress this tick.
    self._buf_stream_ticks = 0
    self._buf_stall_ticks += 1

    if not self._buf_started:
        # Stream has never produced a frame yet.
        if self._buf_stall_ticks < 8:
            self._set_buf_state("connecting")
        else:
            self._set_buf_state("nosignal")
```

```

else:
    # Was playing and has now stalled.
    if self._buf_stall_ticks < 2:
        pass # debounce: brief glitch, don't react yet
    elif self._buf_stall_ticks < 8:
        self._set_buf_state("buffering", self.player.cache_pct)
    else:
        self._set_buf_state("lost")

def _poll_vlc(self):
    """Drive the health dot for embedded VLC from its RC playback clock.
    VLC exposes no buffer-cushion metric, so no seconds are shown."""
    mon = self._vlc_monitor
    if mon is None:
        return
    # User paused on purpose – not a stall.
    if self._vlc_paused:
        self._buf_stall_ticks = 0
        return
    mon.request()
    t = float(mon.time)
    if t > self._buf_last_pos + 0.5:
        self._buf_started = True
        self._buf_stall_ticks = 0
        self._buf_last_pos = t
        self._set_buf_state("stream", show_secs=False)
        return
    self._buf_last_pos = t
    self._buf_stall_ticks += 1
    if not self._buf_started:
        self._set_buf_state(
            "connecting" if self._buf_stall_ticks < 8 else "nosignal",
            show_secs=False)
    elif self._buf_stall_ticks < 2:
        pass
    elif self._buf_stall_ticks < 8:
        self._set_buf_state("buffering", show_secs=False)
    else:
        self._set_buf_state("lost", show_secs=False)

def _update_channel_fail_icon(self, stream_id: str, is_fail: bool) -> None:
    for i in range(self.ch_list.count()):
        item = self.ch_list.item(i)
        if item:
            s = item.data(Qt.ItemDataRole.UserRole)
            if s and str(s.get("stream_id", "")) == stream_id:
                item.setData(FAIL_ROLE, is_fail)
                break
    self.ch_list.viewport().update()

def _set_buf_state(self, state: str, pct: int = 0, show_secs: bool = True):
    # Record failure on first transition into nosignal or lost.
    if state in ("nosignal", "lost") and not self._failure_recorded:
        sid = str((self._playing_stream or {}).get("stream_id", ""))
        if sid:
            self._failure_recorded = True
            self.config.record_stream_failure(sid)
            self._update_channel_fail_icon(sid, self.config.is_stream_unreliable(sid))
        secs = f" · {self.player.cache_secs}s" if show_secs else ""
        if state == "stream":

```



```

        text = f"● STREAM{secs}"
        color, bg, border = "#00c853", "#071a0d", "#1a4f2a"
    elif state == "connecting":
        text = "● CONNECTING"
        color, bg, border = "#42a5f5", "#071018", "#1a3a5a"
    elif state == "buffering":
        text = f"● BUFFERING{secs}"
        color, bg, border = "#ffb300", "#1a1400", "#4f3c00"
    elif state == "nosignal":
        text = "● NO SIGNAL"
        color, bg, border = "#f44336", "#1a0707", "#4f1c1c"
    else: # lost
        text = "● LOST"
        color, bg, border = "#f44336", "#1a0707", "#4f1c1c"
    self._buf_indicator.setText(text)
    self._buf_indicator.setStyleSheet(
        f"color:{color}; font-size:10px; font-weight:bold;"
        f" background:{bg}; border:1px solid {border};"
        f" border-radius:4px; padding:0 6px;"
    )
    self._buf_indicator.show()

# — global search —————

def _open_global_search(self):
    if not self.api:
        self._status("Connect to a provider first")
        return
    dlg = _GlobalSearchDialog(self.api, parent=self)
    dlg.setStyleSheet(self.styleSheet())
    dlg.stream_selected.connect(self._on_search_play)
    dlg.exec()

def _on_search_play(self, s: dict, url: str):
    self._stack.setCurrentIndex(1)
    self._on_tab_changed(0)
    self._play(s, url)

# — continue watching helpers —————

def _show_continue_watching(self):
    self._img_queue.clear()
    self._epg_queue.clear()
    self._season_queue.clear()
    self._vod_q_queue.clear()
    self._live_q_queue.clear()
    self._load_gen += 1
    gen = self._load_gen

    prefix = "vod:" if self._tab == 1 else "ep:"
    entries = [
        (k, v) for k, v in self.config.watch_progress.items()
        if k.startswith(prefix)
    ]
    entries.sort(key=lambda x: x[1].get("ts", 0), reverse=True)

    self.poster_grid.clear()
    if not entries:
        _placeholder(self.poster_grid, "Nothing to resume yet")
    return

```

```

self._ch_header.setText(f"CONTINUE WATCHING  ({len(entries)})")
blank = QIcon(self._blank_poster())
self.poster_grid.setUpdatesEnabled(False)

for key, data in entries:
    s = dict(data.get("stream") or {})
    s["_watch_key"] = key

    item = QListWidgetItem("")
    item.setData(Qt.ItemDataRole.UserRole, s)
    item.setData(WATCH_PROG_ROLE, data)

    url = s.get("stream_icon") or s.get("cover") or ""
    if url:
        item.setIcon(blank)
    else:
        name = s.get("name") or data.get("name", "")
        gp = self._make_generic_poster(name)
        item.setData(BASE_POSTER_ROLE, gp)
        self._refresh_poster_icon(item, gen)

    self.poster_grid.addItem(item)

self.poster_grid.setUpdatesEnabled(True)

for i, (key, data) in enumerate(entries):
    s = data.get("stream") or {}
    url = s.get("stream_icon") or s.get("cover") or ""
    name = s.get("name") or data.get("name", "")
    if url:
        self._fetch_img(i, url, gen, _POSTER_W, _POSTER_H, name, "")

self._flush_img_queue()

# Queue quality badges – mirrors the regular movie/serie loading paths
if self._tab == 1:
    self._vod_q_queue = [
        (i, str((data.get("stream") or {}).get("stream_id", ""))))
        for i, (key, data) in enumerate(entries)
        if (data.get("stream") or {}).get("stream_id")
    ]
    self._flush_vod_q_queue(gen)
elif self._tab == 2:
    self._season_queue = [
        (i, str((data.get("stream") or {}).get("series_id", ""))))
        for i, (key, data) in enumerate(entries)
        if (data.get("stream") or {}).get("series_id")
    ]
    self._flush_season_queue(gen)

def _save_progress(self):
    if not self._current_vod_key or not self._playing_stream:
        return
    pos = self.player.time_pos
    if pos > 10:
        import time
        dur = self.player.duration
        self.config.watch_progress[self._current_vod_key] = {
            "pos": pos,

```

```

        "name": self._playing_stream.get("name", ""),
        "ts": int(time.time()),
        "dur": dur if dur and dur > 0 else None,
        "stream": self._playing_stream,
    }
    self.config.save()

def _on_volume(self, v: int):
    self.config.volume = v
    if self._ext_proc is not None and self._vlc_embedded:
        try:
            if self._ext_proc.poll() is None and self._ext_proc.stdin:
                vlc_vol = int(v / 130 * 512)
                self._ext_proc.stdin.write(f"volume {vlc_vol}\n".encode())
                self._ext_proc.stdin.flush()
            except Exception:
                pass
        return
    self.player.set_volume(v)

# — loading spinner —————

_SPIN_FRAMES = ""

def _make_spinner(self, parent: QWidget) -> QLabel:
    lbl = QLabel("", parent)
    lbl.setStyleSheet("color:#42a5f5; font-size:24px; background:transparent;")
    lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
    lbl.setFixedSize(40, 40)
    lbl.hide()
    parent.installEventFilter(self)
    self._spin_widgets.append(lbl)
    return lbl

def eventFilter(self, obj, event):
    if event.type() == event.Type.Resize:
        for lbl in self._spin_widgets:
            if lbl.parent() is obj:
                lbl.move((obj.width() - lbl.width()) // 2,
                        (obj.height() - lbl.height()) // 2)
    return super().eventFilter(obj, event)

def _show_spinner(self, lbl: QLabel):
    parent = lbl.parent()
    lbl.move((parent.width() - lbl.width()) // 2,
            (parent.height() - lbl.height()) // 2)
    lbl.raise_()
    lbl.show()
    if not self._spin_timer.isActive():
        self._spin_timer.start()

def _hide_spinner(self, lbl: QLabel):
    lbl.hide()
    if not any(l.isVisible() for l in self._spin_widgets):
        self._spin_timer.stop()

def _tick_spinners(self):
    self._spin_idx = (self._spin_idx + 1) % len(self._SPIN_FRAMES)
    ch = self._SPIN_FRAMES[self._spin_idx]
    for lbl in self._spin_widgets:

```

```
        if lbl.isVisible():
            lbl.setText(ch)

# -----

def _update_topbar_time(self):
    self._topbar_time.setText(datetime.now().strftime("%I:%M %p  %b %d"))

def _show_log(self):
    self._log_panel.show(); self._log_panel.raise_()

def _status(self, msg: str):
    self.statusBar().showMessage(msg, 6000)

def closeEvent(self, event):
    self._save_progress()
    if self._ext_proc is not None:
        try:
            if self._ext_proc.poll() is None:
                self._ext_proc.terminate()
        except Exception:
            pass
    self.player.terminate_mpv()
    self.config.save()
    super().closeEvent(event)

# --- Series episode browser -----

class _EpDelegate(QStyledItemDelegate):
    """Draws a numbered blue badge on the left of each episode row."""

    _BADGE = 28

    def sizeHint(self, option, index):
        return QSize(option.rect.width(), 48)

    def paint(self, painter, option, index):
        painter.save()
        r = option.rect

        # row background
        bg = QColor("#1a2535") if option.state & QStyle.StateFlag.State_Selected else QColor("#111827")
        painter.fillRect(r, bg)

        # subtle bottom separator
        painter.setPen(QPen(QColor("#1e2a3a"), 1))
        painter.drawLine(r.left() + 56, r.bottom(), r.right(), r.bottom())

        # numbered badge
        ep_num = index.data(Qt.ItemDataRole.UserRole + 1) or ""
        badge_x = r.left() + 12
        badge_y = r.top() + (r.height() - self._BADGE) // 2
        badge_r = self._BADGE
        painter.setBrush(QColor("#1565c0"))
        painter.setPen(Qt.PenStyle.NoPen)
        painter.drawRoundedRect(badge_x, badge_y, badge_r, badge_r, 6, 6)
        painter.setPen(QColor("white"))
        f = painter.font()
        f.setPixelSize(12)
```

```

        f.setBold(True)
        painter.setFont(f)
        painter.drawText(badge_x, badge_y, badge_r, badge_r,
                        Qt.AlignmentFlag.AlignCenter, str(ep_num))

    # episode title
    text = index.data(Qt.ItemDataRole.DisplayRole) or ""
    text_x = badge_x + badge_r + 12
    painter.setPen(QColor("#d0d8e8"))
    f2 = painter.font()
    f2.setPixelSize(13)
    f2.setBold(False)
    painter.setFont(f2)
    text_rect = r.adjusted(text_x - r.left(), 0, -12, 0)
    painter.drawText(text_rect, Qt.AlignmentFlag.AlignVCenter | Qt.TextFlag.TextSingleLine, text)

    painter.restore()

class SeriesDialog(QDialog):
    """Android-style season/episode picker for a series."""

    def __init__(self, title: str, episodes_by_season: dict, api,
                 series_info: dict | None = None, cover_url: str = "", parent=None):
        super().__init__(parent)
        self.setWindowTitle(title)
        self.setMinimumSize(720, 560)
        self._api = api
        self._eps = episodes_by_season
        self._info = series_info or {}
        self._cover_url = cover_url
        self._seasons = sorted(episodes_by_season.keys(),
                               key=lambda x: int(x) if str(x).isdigit() else 0)
        self._season_btns: list[QPushButton] = []
        self._selected_url: str | None = None
        self._selected_name: str | None = None
        self._build()
        if self._seasons:
            self._select_season(0)
        if cover_url:
            self._load_cover(cover_url)

    # — build —————

    def _build(self):
        vl = QVBoxLayout(self)
        vl.setContentsMargins(0, 0, 0, 0)
        vl.setSpacing(0)

        # — header bar —————
        hdr = QWidget()
        hdr.setFixedHeight(48)
        hdr.setStyleSheet(f"background:{_BG}; border-bottom:1px solid {_BORDER};")
        hl = QHBoxLayout(hdr)
        hl.setContentsMargins(12, 0, 12, 0)
        close_btn = QPushButton("-")
        close_btn.setObjectName("icon_btn")
        close_btn.setFixedSize(36, 36)
        close_btn.setStyleSheet(
            "QPushButton{color:#d0d8e8;font-size:18px;background:transparent;border:none;}")

```

```

        "QPushButton: hover{color:white;}"
    )
    close_btn.clicked.connect(self.reject)
    hl.addWidget(close_btn)
    title_lbl = QLabel(self.windowTitle())
    title_lbl.setStyleSheet("color:white;font-size:15px;font-weight:bold;")
    title_lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
    hl.addWidget(title_lbl, stretch=1)
    # placeholder to balance the close button
    spacer = QWidget()
    spacer.setFixedSize(36, 36)
    hl.addWidget(spacer)
    vl.addWidget(hdr)

    # — series info panel —————
    info_panel = QWidget()
    info_panel.setFixedHeight(145)
    info_panel.setStyleSheet(f"background:{_PANEL};border-bottom:1px solid {_BORDER};")
    il = QHBoxLayout(info_panel)
    il.setContentsMargins(16, 10, 16, 10)
    il.setSpacing(14)

    self._cover_lbl = QLabel()
    self._cover_lbl.setFixedSize(75, 115)
    self._cover_lbl.setStyleSheet("background:#1a2535;border-radius:4px;")
    self._cover_lbl.setAlignment(Qt.AlignmentFlag.AlignCenter)
    il.addWidget(self._cover_lbl)

    text_col = QVBoxLayout()
    text_col.setSpacing(4)
    name_lbl = QLabel(self.windowTitle())
    name_lbl.setStyleSheet("color:white;font-size:14px;font-weight:bold;background:transparent;")
    name_lbl.setWordWrap(False)
    text_col.addWidget(name_lbl)

    overview = (self._info.get("plot") or self._info.get("description")
                or self._info.get("overview") or "")
    if overview:
        desc_lbl = QLabel(overview)
        desc_lbl.setStyleSheet(f"color:{_DIM};font-size:13px;background:transparent;")
        desc_lbl.setWordWrap(True)
        desc_lbl.setMaximumHeight(80)
        text_col.addWidget(desc_lbl)
    text_col.addStretch()
    il.addLayout(text_col, stretch=1)
    vl.addWidget(info_panel)

    # — season tab bar —————
    tab_outer = QWidget()
    tab_outer.setFixedHeight(52)
    tab_outer.setStyleSheet(f"background:{_BG};border-bottom:1px solid {_BORDER};")
    tab_outer_l = QHBoxLayout(tab_outer)
    tab_outer_l.setContentsMargins(0, 0, 0, 0)

    scroll = QScrollArea()
    scroll.setHorizontalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAsNeeded)
    scroll.setVerticalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOff)
    scroll.setWidgetResizable(True)
    scroll.setFrameShape(QFrame.Shape.NoFrame)
    scroll.setStyleSheet("background:transparent;")

```

```

scroll.horizontalScrollBar().setStyleSheet(
    "QScrollBar:horizontal{height:4px;background:#1a2535;border-radius:2px;}"
    "QScrollBar::handle:horizontal{background:#1565c0;border-radius:2px;}"
    "QScrollBar::add-line:horizontal,QScrollBar::sub-line:horizontal{width:0;}"
)

tab_widget = QWidget()
tab_widget.setStyleSheet("background:transparent;")
self._tab_row = QHBoxLayout(tab_widget)
self._tab_row.setContentsMargins(12, 8, 12, 8)
self._tab_row.setSpacing(8)
self._tab_row.addStretch()

for i, s in enumerate(self._seasons):
    btn = QPushButton(f"Season {s}")
    btn.setFixedHeight(32)
    btn.setCheckable(True)
    btn.setCursor(Qt.CursorShape.PointingHandCursor)
    btn.clicked.connect(lambda _checked, idx=i: self._select_season(idx))
    self._tab_row.insertWidget(self._tab_row.count() - 1, btn)
    self._season_btns.append(btn)

scroll.setWidget(tab_widget)
tab_outer_l.addWidget(scroll)
vl.addWidget(tab_outer)

# — episode list —————
self._ep_list = QListWidget()
self._ep_list.setItemDelegate(_EpDelegate(self._ep_list))
self._ep_list.setStyleSheet(
    f"QListWidget{{background:{_BG};border:none;outline:none;}}"
    f"QListWidget::item{{border:none;padding:0;}}"
)
self._ep_list.itemClicked.connect(self._confirm)
vl.addWidget(self._ep_list, stretch=1)

def _season_btn_style(self, selected: bool) -> str:
    if selected:
        return (
            f"QPushButton{{background:{_SEL};color:white;border:none;"
            f"border-radius:6px;font-size:12px;padding:0 14px;}}"
            f"QPushButton:hover{{background:#1976d2;}}"
        )
    return (
        f"QPushButton{{background:transparent;color:{_DIM};"
        f"border:1px solid {_BORDER};border-radius:6px;"
        f"font-size:12px;padding:0 14px;}}"
        f"QPushButton:hover{{color:{_TEXT};border-color:#4a6a8a;}}"
    )

# — slots —————

def _select_season(self, idx: int):
    for i, btn in enumerate(self._season_btns):
        btn.setChecked(i == idx)
        btn.setStyleSheet(self._season_btn_style(i == idx))
    self._current_season_idx = idx
    key = self._seasons[idx]
    self._ep_list.clear()
    for ep in sorted(self._eps.get(key, []), key=lambda e: e.get("episode_num", 0)):

```

```

        num = ep.get("episode_num", "?")
        t = ep.get("title") or f"Episode {num}"
        text = f"{self.windowTitle()} - S{int(self._seasons[idx]):02d}E{int(num):02d} - {t}" \
            if str(self._seasons[idx]).isdigit() and str(num).isdigit() else t
        item = QListWidgetItem(text)
        item.setData(Qt.ItemDataRole.UserRole, ep)
        item.setData(Qt.ItemDataRole.UserRole + 1, num)
        self._ep_list.addItem(item)

def _load_cover(self, url: str):
    import threading, requests as _req
    def _fetch():
        try:
            r = _req.get(url, timeout=8)
            r.raise_for_status()
            return r.content
        except Exception:
            return None
    def _apply(data):
        if not data:
            return
        px = QPixmap()
        px.loadFromData(data)
        if not px.isNull():
            px = px.scaled(75, 115, Qt.AspectRatioMode.KeepAspectRatio,
                           Qt.TransformationMode.SmoothTransformation)
            self._cover_lbl.setPixmap(px)
    w = Worker(_fetch)
    w.signals.result.connect(_apply)
    QThreadPool.globalInstance().start(w)

def _confirm(self, item: QListWidgetItem):
    ep = item.data(Qt.ItemDataRole.UserRole)
    if ep:
        eid = ep.get("id")
        ext = ep.get("container_extension", "mp4")
        self._selected_url = self._api.series_episode_url(eid, ext)
        self._selected_name = item.text()
        self._selected_ep_id = eid
        self._selected_ep_ext = ext
        self._selected_season_idx = self._current_season_idx
        self._selected_ep_row = self._ep_list.currentRow()
        self._seasons_list = self._seasons
        self.accept()

# — properties —————

@property
def selected_url(self) -> str | None:
    return self._selected_url

@property
def selected_name(self) -> str | None:
    return self._selected_name

@property
def selected_season_idx(self) -> int:
    return getattr(self, "_selected_season_idx", 0)

@property

```



```
def selected_ep_row(self) -> int:
    return getattr(self, "_selected_ep_row", 0)

@property
def seasons_list(self) -> list:
    return getattr(self, "_seasons_list", [])

# — helpers —————

def _placeholder(lst: QListWidget, text: str):
    item = QListWidgetItem(text)
    item.setFlags(Qt.ItemFlag.NoItemFlags)
    item.setForeground(Qt.GlobalColor.darkGray)
    lst.addItem(item)

def _decode_title(raw: str) -> str:
    if not raw:
        return ""
    try:
        return base64.b64decode(raw + "==").decode("utf-8")
    except Exception:
        return raw

def _fmt_ts(ts) -> str:
    if not ts:
        return ""
    try:
        return datetime.fromtimestamp(int(ts)).strftime("%H:%M")
    except Exception:
        return ""

def _fmt_secs(seconds: float) -> str:
    s = int(seconds)
    h, rem = divmod(s, 3600)
    m, sec = divmod(rem, 60)
    return f"{h}:{m:02d}:{sec:02d}" if h else f"{m}:{sec:02d}"

def _wrap_text(text: str, fm, max_w: int) -> tuple[str, str]:
    if fm.horizontalAdvance(text) <= max_w:
        return text, ""
    words = text.split()
    line1 = ""
    for i, word in enumerate(words):
        cand = (line1 + " " + word).strip()
        if fm.horizontalAdvance(cand) > max_w:
            rest = " ".join(words[i:])
            if fm.horizontalAdvance(rest) > max_w:
                rest = fm.elidedText(rest, Qt.TextElideMode.ElideRight, max_w)
            return line1 or cand, rest
        line1 = cand
    return line1, ""
```