

Docker & Media Library

- [A menu script for Docker media containers, with a VPN routing check](#)
- [Clean up leftover {tmdb-...} folders in a media library](#)
- [Remove movie folders that have no video files \(safely\)](#)

A menu script for Docker media containers, with a VPN routing check

If you run a group of media containers (the *arr apps, download clients, flaresolvert and so on), a small menu script saves typing for the common jobs. The most useful part is a check that download clients really leave through the VPN.

Pick the "media" group by name

```
media_containers() {  
  docker ps --format '{{.Names}}' |  
    grep -E 'sonarr|radarr|lidarr|prowlarr|bazarr|sabnzbd|qbittorrent|deluge|transmission|flaresolvert' | sort  
}
```

A simple menu

```
PS3="Choose: "  
select opt in "List containers" "Restart media group" "Check VPN routing" "Quit"; do  
  case $REPLY in  
    1) docker ps --format 'table {{.Names}}\t{{.Status}}' ;;  
    2) read -rp "Restart $(media_containers | wc -l) containers? (y/N) " a  
       [[ $a == [yY] ]] && media_containers | xargs -r docker restart ;;  
    3) vpn_check ;;  
    4) break ;;  
  esac  
done
```

Ask for confirmation before bulk actions. A "restart all" that also **starts stopped containers** will bring back anything you stopped on purpose.

Check that download clients use the VPN

Compare the public IP seen from inside each client with the host's own public IP. They should be **different**.

```
vpn_check() {
  local host_ip c ip
  host_ip=$(curl -fsS https://ifconfig.me)
  for c in qbittorrent deluge transmission sabnzbd; do
    docker inspect "$c" >/dev/null 2>&1 || continue
    ip=$(docker exec "$c" curl -fsS --max-time 10 https://ifconfig.me 2>/dev/null) || ip="(no answer)"
    if [[ $ip == "$host_ip" ]]; then echo "LEAK $c: $ip (same as host)"
    else echo "OK $c: $ip"; fi
  done
}
```

- Many images don't include `curl`. Use `wget -qO-`, or run the check from the VPN container (gluetun) that the clients share.
- Inside `[[ ... ]]`, put spaces before `]]`. Writing `"$ip"]` is a syntax error that only shows up when that branch runs.

Small bash pitfalls in scripts like this

- Send status messages to **stderr** (`>&2`) in functions whose stdout you capture with `$(...)`.
- Counters set inside `cmd | while ...` are lost, because the loop runs in a subshell. Use `while ...; done <<(cmd)` instead.
- `local x=$(cmd); echo $?` always reports 0. Declare the variable first: `local x; x=$(cmd)`.

Clean up leftover {tmdb-...} folders in a media library

Media managers (Radarr, Sonarr, Jellyfin) name folders like `Movie (2020) {tmdb-12345}`. When an item is removed or re-imported, a folder can be left behind with just artwork or an `.nfo` file. This how-to finds those folders and removes them, after showing you the list first.

Find candidates (dry run)

A candidate is a folder with `{` or `}` in its name, no subfolders, exactly two files, and no video file:

```
#!/bin/bash
root=${1:?usage: tmdb-clean <library> [--delete]}
find "$root" -mindepth 1 -type d \( -name '*{' -o -name '*}' \) -print0 |
while IFS= read -r -d '' d; do
    subs=$(find "$d" -mindepth 1 -maxdepth 1 -type d | wc -l)
    files=$(find "$d" -maxdepth 1 -type f | wc -l)
    vids=$(find "$d" -maxdepth 1 -type f \( -iname '*.mkv' -o -iname '*.mp4' -o -iname '*.avi' -o -iname '*.m4v' \) |
wc -l)
    if (( subs == 0 && files == 2 && vids == 0 )); then
        if [[ ${2:-} == --delete ]]; then rm -rf -- "$d"; echo "deleted: $d"
        else echo "would delete: $d"; fi
    fi
done
```

Use it

```
./tmdb-clean.sh /mnt/media/movies      # review the list
./tmdb-clean.sh /mnt/media/movies --delete # then delete
```

- The **video check** is important. Without it, a folder with one movie and one subtitle file also matches "exactly two files".

- `-print0` with `read -d ''` handles any file name safely.
- Rescan the library in your media server afterwards.

Remove movie folders that have no video files (safely)

Over time a movie library collects folders that hold only posters, `.nfo` files or subtitles, because the movie itself was deleted or moved. This finds them, shows the list, and deletes only after you confirm.

The logic

A folder is a candidate when **nothing under it** is a video file. Never include the library root itself (`-mindepth 1`).

```
#!/bin/bash
set -euo pipefail
root=${1:?usage: clean-empty-movies <library> [--delete]}
has_video() {
    find "$1" -type f \( -iname '*.mkv' -o -iname '*.mp4' -o -iname '*.m4v' -o -iname '*.avi' -o -iname '*.wmv' \) \
        -print -quit | grep -q .
}
mapfile -d '' dirs < <(find "$root" -mindepth 1 -depth -type d -print0)
todo=()
for d in "${dirs[@]}"; do
    has_video "$d" || todo+=("$d")
done
printf '%s\n' "${todo[@]}"
[[ ${2:-} == --delete ]] || { echo "Dry run. Re-run with --delete."; exit 0; }
read -rp "Delete ${#todo[@]} folders? (y/N) " a
[[ $a == [yY] ]] || exit 0
for d in "${todo[@]}"; do rm -rf -- "$d"; done
```

Key points

- **Get the test the right way round.** `has_video` returns success (0) when a folder *has* video, and you delete when it does **not**. If you invert this by mistake, the list holds every real movie folder. That's why the script defaults to a dry run; always read the list.
- `-print -quit` stops at the first video found, which is fast on big libraries.
- Subtitle-only folders (`.srt`) count as "no video" here. Adjust the pattern if you keep stray subtitles on purpose.
- Rescan the library in Plex or Jellyfin afterwards.