



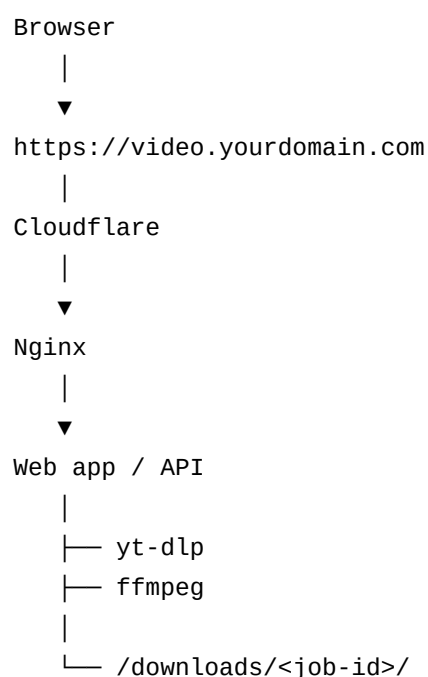
Today 5:02 AM

I don't need to one-shot it, rather I want to do it in a decent way, but I want to develop my own website using <https://github.com/yt-dlp/yt-dlp>. Why? Because I do work on live news broadcast and sometimes we do show videos from different sources, but I want to show the video is HQ, so I try to download them and then add the source when shown on live broadcast. But yeah, the sites out there are full with ads and working 50/50. I do have a domain, a profile on cloudflare and a remote Ubuntu VPS, so I'm set in terms of infrastructure.

Worked for 19s >

Yeah. For your use case, this is a pretty sensible project, and I would **not** start by trying to build some polished public downloader. I'd treat it as an internal newsroom utility running on your VPS.

The architecture I'd recommend is:



The actual workflow could be:

1. Paste a URL.
2. Backend asks `yt-dlp` for metadata **without downloading anything**.
3. Website shows:
 - title
 - source/site
 - duration
 - thumbnail
 - available resolutions/codecs
4. You select something like:
 - **Best available quality**
 - **1080p**
 - **Broadcast-friendly MP4**
5. Backend starts the download.
6. Frontend shows progress.
7. When finished, you get:
 - Download video
 - original source URL
 - uploader/account
 - title
 - maybe upload date
8. Server automatically deletes the video after, say, 1 hour.

That last metadata part would actually be useful for your workflow because you could copy the source attribution directly rather than manually returning to the original page.

Stack

I would keep it very boring:

Frontend:

HTML + CSS + small amount of JavaScript

Backend:

Python + FastAPI

Downloader:

yt-dlp Python API

Media:

ffmpeg

Reverse proxy:

Nginx

Deployment:

Docker Compose

Protection:

Cloudflare + authentication

You don't need React, Next.js, Redis, PostgreSQL, Kubernetes or anything like that initially.

yt-dlp is explicitly usable as a Python library, so you don't have to build everything around spawning shell commands. It also provides structured JSON metadata and extensive format selection. Its current default format selector is effectively `bv*+ba/b`, meaning it normally tries to obtain high-quality video/audio streams and merge them when appropriate.  [GitHub](#)

That means your backend can conceptually do:

```
Python                                                                    Run

with yt_dlp.YoutubeDL(options) as ydl:
    info = ydl.extract_info(url, download=False)
```

and first obtain information about the video.

Then another endpoint starts the actual download.

One thing I'd design differently from normal yt-dlp websites

Don't equate "highest quality" with "best file for television."

You might encounter:

```
2160p AV1 + Opus WebM
```

which can be technically excellent quality but annoying to put into a broadcast/editing workflow.

You could therefore have two main buttons:

BEST QUALITY

Keeps the highest-quality streams yt-dlp can obtain.

BROADCAST / EDIT FRIENDLY

Prefer H.264 + AAC + MP4.

Maybe later:

Best available

2160p

1080p

720p

Container:

Auto

MP4

MOV

Codec preference:

Auto

H.264

If conversion is necessary, **ffmpeg** handles that part. yt-dlp itself already integrates closely with ffmpeg for merging and post-processing.  [GitHub](#)

There is an important performance difference, though:

Download AV1 → AV1

fast / little CPU

Download separate H.264 + AAC → merge MP4

very fast / little CPU

Download AV1 → transcode H.264

slow / heavy CPU

So ideally we first choose compatible source streams rather than transcoding everything.

I would also make downloads job-based

Don't have this happen:

```
http
POST /download
|
└── HTTP connection sits open for 4 minutes
```

Instead:

```
http
POST /api/jobs
{
  "url": "...
}

↓

{
  "job_id": "a824d9"
}
```

Then:

```
GET /api/jobs/a824d9
```

might return:

JSON

```
{
  "status": "downloading",
  "progress": 67,
  "speed": "14.2 MiB/s",
  "eta": 8
}
```

Eventually:

JSON

```
{
  "status": "complete",
  "filename": "video.mp4"
}
```

This also avoids tying the website's HTTP request lifetime to however long TikTok/YouTube/X/Instagram takes to respond.

yt-dlp provides progress hooks when embedded in Python, so getting this information is straightforward.  [GitHub](#)

Security is actually an interesting part of this project

Because **the user controls a URL that your VPS will fetch**, I would treat this endpoint somewhat like an SSRF boundary.

Even if you're the only user initially, I'd build a few restrictions from the start:

- ✓ only http:// and https:// input
- ✓ reject localhost
- ✓ reject 127.0.0.0/8
- ✓ reject RFC1918 networks
- ✓ reject 169.254.0.0/16
- ✓ reject IPv6 loopback/link-local/private destinations
- ✓ impose maximum video duration
- ✓ impose maximum output size
- ✓ timeout downloads
- ✓ randomized server-side filenames
- ✓ never put supplied titles directly into shell commands
- ✓ never run yt-dlp as root
- ✓ dedicated downloads directory

And critically:

Python	Run
<pre>subprocess.run(["yt-dlp", url], shell=False)</pre>	

rather than constructing:

Python	Run
<pre>os.system("yt-dlp " + url)</pre>	

Although I'd still prefer the Python API.

This isn't theoretical security paranoia either. yt-dlp had several security fixes in its June/July 2026 releases, including issues involving command-template injection, dangerous output file types and aria2c manifest handling. The current latest release I found is **2026.07.04**, and the project now recommends Python 3.11+.  [GitHub](#)

So your container should make updating yt-dlp very easy.

Authentication will matter

I wouldn't expose:

```
download.packet.garden
```

to everybody.

Otherwise you have effectively created a free bandwidth/transcoding service on your VPS.

For something used by you at work, I'd probably put **Cloudflare Access** in front of it.

Then:

```
Internet
|
Cloudflare Access
|
your authenticated account
|
Nginx
|
application
```

No custom user database or login system necessary.

If later a couple of coworkers use it, you can authorize their accounts too.

Cookies can come later

This is one part I would deliberately **not solve during version 1**.

Some sources will work anonymously. Others—particularly Instagram, age-restricted YouTube videos, private material, etc.—may require authentication cookies.

yt-dlp supports browser cookies and cookie files, and its documentation specifically recommends browser-cookie extraction in appropriate cases.  [GitHub](#) +1

But storing your authenticated YouTube/Instagram/etc. sessions on an Internet VPS creates another security problem.

So:

```
v1
publicly accessible videos only

v2
carefully designed cookie/account support
```

would be my approach.

And one more thing I think is worth building because this is for **actual news production** rather than casual downloading:

SOURCE

Reuters / YouTube / X / TikTok / Instagram

Original URL:

https://...

Uploader:

...

Title:

...

Published:

...

Downloaded:

2026-08-11 05:xx

Resolution:

1920x1080

Codec:

H.264

FPS:

50

Duration:

00:01:23

You could even store that as a tiny `.json` sidecar next to the media file.

Then the tool becomes less of a **"video downloader website"** and more of a little **media-ingest tool**, which fits your work considerably better.

For a first milestone, I'd build only:

Paste URL
↓
Inspect metadata
↓
Show available quality
↓
Download
↓
Progress bar
↓
Download MP4
↓
automatic cleanup

Once that reliably works on the VPS, we can add source metadata, broadcast-friendly format presets, authentication/cookies and other sites.

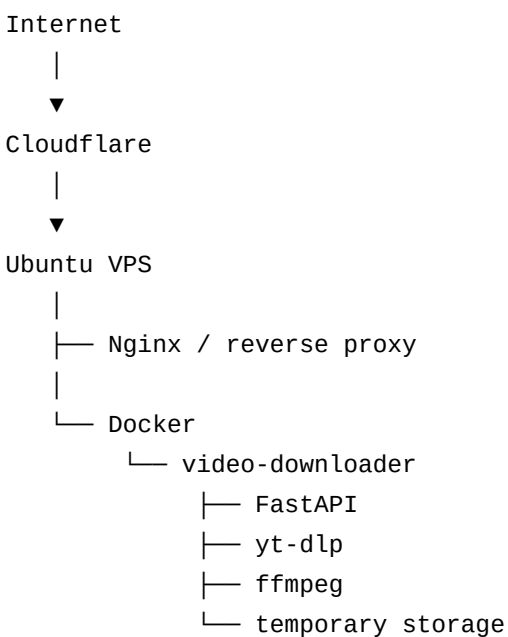
Given your existing Ubuntu VPS + domain + Cloudflare setup, you already have basically all the infrastructure needed for this.

    ...  Sources

Yeah, I can't really build this as an internal tool because I'm not related to IT at my work, and I don't also have any friends in IT at my work. And also, I know that they will not be willing to do anything regarding that because I have asked about internal tool deployment before. So then that's why I have come down to just deploying it on my own VPS and using my own domain. But yeah, since I really use it mostly alone, then I would just rather use a login account because I do, I have to do it daily, but it's not like I have to do it all day. So I don't really mind to log in. I rather just have it all secure and safe.

Yeah, that changes the framing a little. I wouldn't consider it an *internal company tool* at all. It's **your privately hosted web application that you happen to use as part of your workflow.**

And if you're essentially the only user, I agree with using authentication. I'd design it like this:



Then something like `video.example.com` presents a login page before any downloader functionality is accessible.

Authentication

Since you're building this partly to learn, there are two reasonable approaches.

Option A — Cloudflare Access

Cloudflare authenticates you before traffic ever reaches the application:



This is probably the strongest/easiest solution for a single-user service because your application doesn't even need to implement passwords.

Option B — build authentication into your application

video.example.com/login

Username

Password

[Login]

Then you'd get some useful web-development/security experience:

- Argon2id password hashing
- secure session cookies
- HttpOnly
- Secure
- SameSite
- CSRF protection where appropriate
- login rate limiting
- session expiration
- logout/session invalidation

For a single account, you don't even need registration, password-reset emails, user management, etc. You can create the account administratively.

I'd actually lean toward **Option B for this project if learning is part of the point**, while keeping Cloudflare in front of the site. It's small enough that implementing authentication properly is manageable.

And I wouldn't one-shot this either

I'd build it progressively.

Phase 1: Get a minimal FastAPI application running locally:

URL: [_____]

Inspect

Submit URL → backend calls yt-dlp → returns title, duration, source, thumbnail and available formats.

No downloading yet.

Phase 2: Implement downloads and ffmpeg merging.

Phase 3: Add job handling and progress reporting.

Downloading...

 81%

18.4 MB/s

ETA: 00:07

Phase 4: Add authentication.

Phase 5: Dockerize it.

Phase 6: deploy to the VPS behind your reverse proxy and Cloudflare.

Phase 7: harden it—resource limits, SSRF considerations, download limits, temporary-file cleanup, logging, rate limiting, etc.

That way, every layer works before we introduce the next one.

And there's no reason the eventual UI needs to look like the sketchy downloader websites you're currently using. Since it's yours, we can make it essentially a clean little **broadcast media ingest interface** optimized around exactly what you need: paste source → inspect → choose quality → download → copy attribution.

When you're ready to actually start, I'd begin with **Phase 1 locally**, including setting up the project structure and understanding exactly how we should interface with yt-dlp rather than dumping an entire application on you at once.

      Sources

ChatGPT can make mistakes. Check important info.