# Self-Hosting Nextcloud and Jellyfin: From Beginner to Master

## Introduction

Imagine having your own cloud storage and media streaming server that you fully control. In this comprehensive guide, we will **set up a self-hosted Nextcloud and Jellyfin server** on a Linux machine from scratch. Nextcloud provides an open-source, self-hosted file sync and collaboration platform (much like Dropbox/Google Drive, but you run it yourself), and Jellyfin is a free, volunteer-built media server that lets you stream your movies, music, and photos to any device, with no strings attached. By the end of this guide, you will have a solid understanding of how all the pieces fit together – from Linux setup, Docker containers, reverse proxies with Nginx, securing connections with Let's Encrypt SSL, firewall configurations, to remote access via Tailscale VPN. We will go step-by-step, **from noob to master**, explaining not only *what* commands to run but also *how and why* each component works. This way, you can replicate the entire setup and confidently answer any interview question about its inner workings.

**What We Will Achieve:** We'll deploy Nextcloud (with a MariaDB database) and Jellyfin inside Docker containers on an Ubuntu server. We'll use Nginx as a reverse proxy to expose Nextcloud over HTTPS with a trusted domain name, secured by Let's Encrypt SSL certificates. Jellyfin will be kept private, accessible securely through Tailscale (a mesh VPN) so that only you (or your devices) can stream your media. Along the journey, we'll cover essential concepts (like Docker, container volumes, reverse proxy, firewall rules, etc.) and best practices so that you **not only follow steps, but truly understand the system**.

## Understanding the Components

Before diving into implementation, let's briefly introduce each major component and concept in our setup, along with key principles you should master:

- **Nextcloud:** An open-source, self-hosted file sync and collaboration platform. It provides similar functionality to cloud services like Dropbox or Google Drive – you can upload files, share them, sync across devices, use calendars, contacts, and even integrate office tools. The big difference is **you host your own data** instead of trusting a third party. Nextcloud has a web-based interface and also desktop/mobile clients for file syncing. Technically, Nextcloud is a PHP application that typically runs with a web server (like Apache or Nginx with PHP-FPM) and uses a database (e.g. MySQL/MariaDB) to store metadata. In our setup, we will use the official Nextcloud Docker image, which comes prepackaged with Apache and PHP, making deployment easier. We'll also use a

MariaDB container as Nextcloud's database. As you progress to mastery, remember that Nextcloud is highly extensible (with apps for Talk, Office, etc.), but at its core it's a **file storage and sync platform with powerful collaboration features**.

- **Jellyfin:** An open-source media server that you can run at home to stream your personal media (movies, TV shows, music, photos, etc.) to various devices. It's an alternative to Plex or Emby, but completely free with no proprietary restrictions. Jellyfin puts you in control of your media: *"Your media, your server, your way."* It allows you to manage and stream your content to web browsers, phones, smart TVs, etc.. We will run Jellyfin in a Docker container as well. It has a web UI for configuration and streaming. Jellyfin doesn't require a separate database (it uses SQLite internally for metadata) but you will need to point it to your media files and it will take care of indexing them. Master tip: **organize your media files** (movies, music, etc.) in directories and mount them into the Jellyfin container, so Jellyfin can read and stream them. We'll cover this in the implementation.

- **Docker Containers:** Docker is a containerization platform that allows us to package applications with all their dependencies into isolated units called *containers*. Unlike virtual machines, containers do **not** bundle a full OS; instead, multiple containers share the host's operating system kernel, making them lightweight and efficient. For example, our Nextcloud and Jellyfin containers will share the host's Linux kernel but run in isolated user-space environments, each with their own filesystem and process space. Docker ensures consistency ("it works on my machine" will work everywhere) and easy deployment. We will use Docker Compose (a tool to define multi-container applications) to configure and run the Nextcloud and MariaDB containers together. Key concepts to master:

    - *Images vs Containers:* An **image** is like a blueprint (immutable snapshot with app and deps), and a **container** is a running instance of that image. We'll be using official images like `nextcloud`, `mariadb`, and `jellyfin/jellyfin`.

    - *Volumes:* Containers' data can be persisted using **volumes** or bind mounts. We don't want our Nextcloud files or database to vanish if a container is recreated, so we'll mount host directories (or Docker volumes) into the containers for persistent storage (e.g. Nextcloud data folder, database data directory, Jellyfin config and media).

    - *Networking:* Docker by default will isolate containers in a private network and we can **publish ports** to allow host access. For example, mapping container port 80 to host port 8080 for Nextcloud. By default, a published port is accessible on *all* network interfaces of the host, but we will see how to restrict that for security. Also, Docker's interaction with system firewalls (UFW) is a subtle point – by default Docker manipulates iptables and can bypass UFW rules. As a master, you should know that if you rely on UFW, you either need to adjust Docker's settings or explicitly bind services to specific interfaces, which we will do for safety.

- **Nginx Reverse Proxy:** Nginx is a high-performance web server that we'll use as a **reverse proxy** in front of Nextcloud. Reverse proxy means Nginx will sit in between the clients and the Nextcloud container: clients will connect to Nginx on standard web ports (80 for HTTP, 443 for HTTPS), and Nginx will forward those requests to the Nextcloud container which is listening on an internal port. Why do this? Several reasons:

    - **TLS/HTTPS Termination:** We want to serve Nextcloud over HTTPS (secure), and we'll obtain a Let's Encrypt SSL certificate for our domain. Nginx will handle the TLS encryption and decryption, so the Nextcloud container can just speak plain HTTP internally. This offloads the SSL from the app and allows using the well-trusted Let's Encrypt cert.

    - **Domain Name Handling:** Nextcloud needs to know what domain it's accessed by (for generating links, etc.). We will have Nginx respond on a specific domain (e.g. `cloud.yourdomain.com`) and Nextcloud will be configured to trust that. If you access Nextcloud directly by IP or an untrusted domain, it will warn about "untrusted domain" (you might have seen this error in testing). We'll fix that by proper configuration.

    - **Flexible Routing:** With Nginx, you could also host multiple services on one machine (virtual hosts, different subdomains or paths) and route to different containers. In our case, we might decide to add a domain for Jellyfin later. (For simplicity, we won't proxy Jellyfin through Nginx in this guide, but it's possible. We'll instead use Tailscale for Jellyfin access).

- We will provide a sample Nginx configuration for Nextcloud. It will listen on port 80 (for the Let's Encrypt certificate challenge and to redirect HTTP to HTTPS) and on port 443 for HTTPS. The config will pass all requests to Nextcloud's container (running on http://127.0.0.1:8080 in our setup). We'll also include recommended settings from Nextcloud's docs (like client_max_body_size for large uploads, and some security headers). As a master, remember that Nginx (unlike Apache) doesn't natively run PHP – in our case, the Nextcloud container runs Apache/PHP, so Nginx is only proxying. If one were using Nextcloud FPM image, Nginx would need fastcgi configs; but with the standalone image, proxying is fine.

- **Let's Encrypt SSL (Certbot):** Let's Encrypt is a certificate authority that provides **free SSL/TLS certificates** so you can serve HTTPS. We'll use **Certbot**, the official Let's Encrypt client, with its Nginx plugin. Certbot will automatically perform the domain validation (ensuring you own the domain by creating a temporary file on port 80 for Let's Encrypt to check) and then obtain a certificate and configure Nginx to use it. This will give us a trusted certificate (green lock in browser) without paying anything. We'll set this up such that Certbot also auto-renews the cert every few months. In an interview, you might be asked how Let's Encrypt works: you should mention the ACME protocol and the challenge types (we'll be using HTTP-01 challenge on port 80, which requires the server to be reachable from the public internet). **Prerequisite:** you need a domain name (e.g. a DDNS or your own domain) pointing to your server's IP, and port 80 (and 443) accessible externally for the setup. If you cannot open ports (e.g. behind strict NAT), there are alternate ways (DNS challenges or using Tailscale's "Funnel" feature), but we assume a typical scenario where you can expose the web server ports.

- **Firewall (UFW on Ubuntu):** A firewall controls network traffic to your server. **UFW (Uncomplicated Firewall)** is a user-friendly interface to Linux iptables commonly used on Ubuntu. We will configure UFW to allow only necessary ports:
    - SSH (port 22) but possibly restricted to your LAN or via Tailscale only, for security.
    - HTTP (80) and HTTPS (443) for Nextcloud (from anywhere, if you want Nextcloud public).
    - Jellyfin's port (8096 by default) – but instead of opening to the world, we will demonstrate how to restrict it. For example, allow it only through the Tailscale interface (so only Tailscale VPN peers can access) and/or only from LAN. In our example, we will close Jellyfin to the open internet for privacy, and access it via Tailscale.
    - We'll also allow the Tailscale VPN service itself. Tailscale works peer-to-peer and usually doesn't require opening ports on the firewall for functionality, but we do need to allow the Tailscale network interface (tailscale0) to handle traffic for allowed services. We'll see that in practice (UFW rule `allow in on tailscale0`).
- One must be aware that Docker and UFW have an interplay: Docker by default will punch through UFW unless configured otherwise. We will mitigate this by binding services to specific interfaces and using UFW rules accordingly. A "master" consideration: for ultimate security, you could configure Docker's daemon to not bypass UFW (setting `"iptables": false` in Docker daemon config) and manage rules yourself, but this is an advanced topic. In our guide, we'll keep Docker's defaults and just be mindful with binding and UFW.
- **Tailscale VPN:** Tailscale is a managed peer-to-peer VPN based on WireGuard that connects your devices in a secure mesh network. When you install Tailscale on your server and your client devices, they all get an IP (in the `100.x.y.z` range by default) and can directly communicate **as if in the same LAN, regardless of their physical network location**. Tailscale handles NAT traversal automatically (punching through firewalls using techniques like UDP hole punching, and relays if necessary) – usually "it

just works" without needing to open ports. In our scenario, we'll use Tailscale to access Jellyfin securely from anywhere, without exposing Jellyfin to the public internet. We'll also show how you could limit Nextcloud's access to Tailscale (if you chose to keep it private) by firewall rules. Tailscale offers features like **MagicDNS** (assigning a stable name like `yourserver.tailnet.ts.net`) and **HTTPS funnel** (a way to expose a service to the internet via Tailscale proxy with automatic Let's Encrypt cert). While we won't rely on funnel (since we'll set up our own domain and Nginx for Nextcloud), it's good to know this exists as an alternative approach. For mastery, understand that Tailscale uses the WireGuard protocol under the hood and authenticates devices through your identity (OAuth via Google, Microsoft, etc. or your own identity provider). It provides a very convenient way to create a VPN without the complexity of managing a traditional VPN server.

With the above concepts in mind, we have a high-level picture: **Users** will access Nextcloud at `https://yourcloud.domain` – the DNS points to your server's IP, Nginx will terminate TLS and reverse proxy to the Nextcloud Docker container which serves the content from its Apache. Users (or just you) will access Jellyfin via either the LAN IP or the Tailscale IP/domain (e.g. a MagicDNS name) on port 8096 – only devices on your Tailscale network (or LAN) can reach it, since we won't expose Jellyfin to the open internet. The server itself will run Docker to manage the app containers; UFW will enforce that only allowed traffic gets through; and Tailscale will ensure you have secure remote access for maintenance (SSH over Tailscale, if desired) or for using Jellyfin on the go.

Below is a summary diagram of the setup for clarity:

*Architecture overview:* The Nextcloud and MariaDB containers run on the server (Docker host). Nginx on the host forwards external HTTPS traffic to Nextcloud. Jellyfin runs in a container and is accessible to clients only via the private networks (LAN or Tailscale VPN), not exposed on the public internet.

*(Diagram: Users connect either through the internet to Nginx (443) for Nextcloud, or through Tailscale/LAN for Jellyfin on port 8096. The server runs Docker containers: Nextcloud (talking to MariaDB) and Jellyfin. UFW firewall allows 80/443 to Nginx (for Nextcloud) globally, but Jellyfin port only via tailscale0 interface or LAN.)*

## Implementation Plan Overview

Now let's outline the **step-by-step implementation plan** we will follow:

1. **Prepare the Server:** Install Ubuntu (or your Linux OS of choice) and perform initial setup (updates, creating a non-root user with sudo if not already, etc.). Configure basic firewall defaults with UFW (deny incoming, allow outgoing by default).

2. **Install Docker Engine:** We will install Docker on the server to run containers. This involves adding Docker's apt repository and installing the `docker-ce` package, plus the `docker-compose-plugin` (or using Docker Compose via `docker compose` command which is now integrated).

3. **Set Up Persistent Storage Directories:** Create directories on the host for persistent data: e.g. `/mnt/data/nextcloud` for Nextcloud files, `/mnt/data/db` for database files, `/mnt/data/jellyfin/config` and `/mnt/data/jellyfin/cache` for Jellyfin config/cache, and a `/mnt/data/media` directory for actual media files (documents, music, videos, etc. that Jellyfin will serve, and possibly to be accessible via Nextcloud as well).

4. **Configure Docker Compose for Nextcloud + MariaDB:** Write a `docker-compose.yml` file that defines two services: **db** (MariaDB 10.6) and **app** (Nextcloud latest). We will specify environment variables for MariaDB (root password, database name, user, password) and for Nextcloud (to connect to the DB, and to set an initial trusted domain). We'll publish Nextcloud's port to the host (but bound to a specific interface like the LAN IP for safety). We'll use Docker volumes/bind mounts for persistence (Nextcloud html data and MariaDB data).

5. **Launch Nextcloud and MariaDB Containers:** Run `docker compose up -d` to start the containers. Verify that they are running (`docker ps`) and listening on the expected ports. At this point, Nextcloud is running but not yet configured – the first time you access it, you'll run through a web-based setup (create admin account, etc.). However, if we provided the `NEXTCLOUD_TRUSTED_DOMAINS` env, it will already trust the host IP or domain we set.

6. **Initial Nextcloud Web Setup:** In a browser, connect to the Nextcloud instance. If you are on the LAN, you could go to `http://<server_ip>:8080` (since we mapped 8080). If remote, you might go through tailscale (e.g. `http://<tailscale-ip>:8080`) temporarily. You'll see Nextcloud's setup screen – choose an admin username & password, and it will ask for database info. Enter the MariaDB credentials we set (database "nextcloud", user "nextcloud", password, host "db" if using Docker's internal network). Complete the setup and you should land in Nextcloud's web interface. *(If connecting via an IP or untrusted domain, Nextcloud might show "Access through untrusted domain" – then you need to add that domain/IP to trusted domains. We already did for one case via env. We will cover how to adjust this in config if needed.)*

**Deploy Jellyfin Container:** We will now run Jellyfin. You can use Docker Compose as well, but to keep it simple we might use a separate compose file or a direct `docker run` command. We'll map port 8096 (Jellyfin's default HTTP port) to the host. If we want Jellyfin only on tailscale or LAN, we can bind it to the LAN IP (like we did for Nextcloud) or just let it bind to all but use firewall rules. We'll mount volumes: one for Jellyfin config, one for cache, and bind-mount our media directory. For example:

```
docker run -d --name jellyfin \
 -p 8096:8096/tcp \
 -v /mnt/data/jellyfin/config:/config \
 -v /mnt/data/jellyfin/cache:/cache \
 -v /mnt/data/media:/media \
 jellyfin/jellyfin:latest
```

7. This pulls the Jellyfin image and runs it. Jellyfin will now be accessible on `http://<server_ip>:8096` (but we haven't opened that port globally on UFW). We'll configure its access soon.

8. **Organize Media Files:** Make sure you populate `/mnt/data/media` with some subfolders (like `movies/`, `music/`, `photos/`, etc., as the user images showed). Jellyfin's web UI will allow you to add media libraries and you'll point them to the `/media` directory (which inside container maps to `/mnt/data/media` on host). For now, creating empty folders is fine. Later, in Jellyfin's UI, you'll set up libraries (e.g. Movies library -> `/media/movies` path).

**Set Up Nginx Reverse Proxy for Nextcloud:** On the host, install Nginx:

```
sudo apt update && sudo apt install -y nginx
```

Nginx by default will run and listen on port 80 (with a default page). We will create an Nginx configuration for our Nextcloud site. For example, create a file `/etc/nginx/sites-available/nextcloud.conf` with something like:

```
 server {
    listen 80;
    server_name yourcloud.example.com;
    # Redirect all HTTP to HTTPS
    return 301 https://$host$request_uri;
}
server {
    listen 443 ssl http2;
    server_name yourcloud.example.com;
    root /var/www/html; # (not used, but required directive)
    # TLS certificates
    ssl_certificate /etc/letsencrypt/live/yourcloud.example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/yourcloud.example.com/privkey.pem;
    # (Cert paths will be filled by Certbot automatically)

    # Recommended security headers (optional hardening)
    add_header Strict-Transport-Security "max-age=15768000; includeSubDomains; preload"
always;
    add_header X-Content-Type-Options "nosniff" always;
    add_header X-Frame-Options "SAMEORIGIN" always;
    add_header X-XSS-Protection "1; mode=block" always;

    client_max_body_size 2G;        # allow large uploads (adjust as needed)
    client_body_timeout 300s;

    location / {
        proxy_pass http://127.0.0.1:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

This config does the following: listens on 80 to redirect to HTTPS, and listens on 443 for the secure site. It assumes we'll have SSL certs in place (the path under `/etc/letsencrypt/live/` – Certbot will manage that soon). The `proxy_pass` points to `127.0.0.1:8080` where our Nextcloud container is listening. We also pass necessary headers so Nextcloud knows the original host and protocol. Save this file and enable it by linking to sites-enabled:

```
sudo ln -s /etc/nginx/sites-available/nextcloud.conf /etc/nginx/sites-enabled/nextcloud.conf
sudo nginx -t  # test configuration
sudo systemctl reload nginx
```

9. At this point, Nginx is set up, but the SSL certificate is not yet there. We will obtain it next. (If you visit `http://yourcloud.example.com` now, you'd likely get a bad gateway or SSL error until cert is ready. That's expected.)

**Obtain Let's Encrypt SSL Certificate:** Ensure that your DNS for `yourcloud.example.com` points to your server's public IP, and port 80 is open to the internet (temporarily allow it if UFW is on). Then run Certbot:

```
sudo apt install -y certbot python3-certbot-nginx
sudo certbot --nginx -d yourcloud.example.com
```

10. Certbot will ask for an email (for urgent notices), agree to Terms, and then perform the challenge. It will automatically edit the Nginx config to include the `ssl_certificate` lines (we already placed them) and it will obtain the cert from Let's Encrypt. If successful, it will say "Congratulations! Your certificate and chain have been saved at /etc/letsencrypt/live/yourcloud.example.com/fullchain.pem" etc. Certbot's Nginx plugin also usually offers to redirect HTTP to HTTPS automatically (we already set that up manually with our config's 301 return, so either approach is fine). After this, your Nextcloud should be accessible at **`https://yourcloud.example.com`** with a green lock. Nginx is now fronting Nextcloud.

    - Test by browsing to `https://yourcloud.example.com`. You should see the Nextcloud login page. If you hadn't completed the setup earlier, you might see the setup page here (you can complete it over HTTPS). Otherwise, log in with the admin account you created.

    - In Nextcloud's settings, you shouldn't see any warnings about "proxy" or "trusted domain" if configured properly. If you do, we need to adjust Nextcloud config (next step).

11. **Configure Nextcloud for Reverse Proxy (Trusted Domains & Proxies):** Nextcloud's config file (`config.php` in the data volume) has parameters for trusted domains and (if behind proxy) trusted proxies and overwrite protocol. We already set `NEXTCLOUD_TRUSTED_DOMAINS` in the container to include at least the IP. We should add our domain as well. There are two ways:

**Option A:** Edit `config.php` inside the Nextcloud container to add entries. For example:

```
 'trusted_domains' =>
   array (
     0 => 'localhost',
     1 => '192.168.0.111',    // your LAN IP if used
     2 => 'yourcloud.example.com',
   ),
'trusted_proxies' =>
   array (
     0 => '127.0.0.1',       // Nginx on host
   ),
'overwriteprotocol' => 'https',
```

These settings mean Nextcloud will trust those hostnames/IPs in the Host header, and will trust the proxy at 127.0.0.1 (Nginx) to tell it the correct client IP and protocol. `'overwriteprotocol' => 'https'` is crucial so that Nextcloud knows to generate URLs as HTTPS (otherwise it might think it's on HTTP and some redirects or link generations could break). Since we used the Nextcloud Docker image, we could also set these via environment variables in docker-compose. Indeed, `TRUSTED_PROXIES` and `OVERWRITEPROTOCOL` can be set as env vars for the container. For example, we could add:

```
environment:
 - TRUSTED_PROXIES=127.0.0.1
 - OVERWRITEPROTOCOL=https
```

- in our compose for Nextcloud. The community confirms that adding `TRUSTED_PROXIES` and `OVERWRITEPROTOCOL=https` in the Docker compose environment fixes the reverse proxy SSL issues.

  If you had not done this, Nextcloud might have shown a warning in its admin panel like "The reverse proxy header configuration is incorrect" or you might experience redirect loops. Now, with these settings, it should be happy. You can verify in Nextcloud settings (or via the OCC command) that `trusted_domains` includes your domain and that `overwriteprotocol` is set. This ensures **flawless explanation** if someone asks "How did you configure Nextcloud behind Nginx and SSL?" – you can say you set trusted domain and the overwrite/proxy settings so Nextcloud knows it's behind HTTPS.

12. **Adjust Firewall Rules (UFW):** At this stage, our services are running and reachable. Now we enforce the desired security with UFW. We want:

Allow SSH (port 22) *only* from specific sources. For example, if you only connect within LAN, do:

```
sudo ufw limit 22/tcp comment 'Allow SSH from anywhere (rate-limited)'
```

(This allows SSH from anywhere but with rate-limiting. Or better, specify your IP/subnet.)
If using Tailscale for SSH, you can actually block port 22 on the public interface and only allow on tailscale interface. For instance:

```
sudo ufw allow in on tailscale0 to any port 22 comment 'SSH via Tailscale'
sudo ufw delete allow 22/tcp   # remove any wide-open SSH rule
```

- This way, port 22 is only accessible if you're on the Tailscale VPN.

Allow HTTP (80) and HTTPS (443) for Nextcloud. Since we have a public Nextcloud:

```
sudo ufw allow 80/tcp comment 'Allow HTTP'
sudo ufw allow 443/tcp comment 'Allow HTTPS'
```

- (If you locked down to tailscale only, you would instead allow on tailscale0 interface or skip opening these globally. But we assume you want Nextcloud available publicly.)

Jellyfin (8096): We do **not** open 8096 to the world. Instead, we allow it only via Tailscale (and/or LAN). If your Jellyfin container is bound to all interfaces, Docker would by default expose it, but recall Docker might bypass UFW. We can enforce with UFW rules:

```
 sudo ufw allow in on tailscale0 to any port 8096 comment 'Allow Jellyfin via Tailscale'
sudo ufw allow from 192.168.0.0/24 to any port 8096 comment 'Allow Jellyfin via LAN'
sudo ufw deny 8096/tcp  # (optional explicit deny on others, though default deny covers it)
```

- This means: Jellyfin's port can be accessed by any Tailscale peer (traffic on tailscale0) and by devices on your local LAN subnet, but not from the general internet. In fact, earlier we saw an example UFW status where a rule like "8096/tcp ALLOW IN on tailscale0" was present, and the allow for LAN might have been removed to strictly funnel through tailscale. You can adjust per your needs.

  **Note:** Because Docker published 8096, you must be aware that Docker's iptables might still allow some access. By binding Jellyfin's container to a specific interface (like `-p 192.168.0.111:8096:8096` in the run command), you can ensure it isn't even listening on the external interface. The screenshots show Nextcloud was bound to `192.168.0.111:8080`. We could similarly bind Jellyfin to `192.168.0.111:8096`. If bound this way, then by default only LAN can see it (and Tailscale, if we allow routing from tailscale to LAN IP). The firewall approach above is another layer of control.

Allow Tailscale traffic: Typically, when you enable UFW after installing Tailscale, Tailscale adds a rule to allow traffic on the tailscale0 interface. You might need:

```
 sudo ufw allow in on tailscale0
```

- This allows **any** Tailscale VPN traffic in (which is fine, because only your authenticated devices are on that network). In UFW status it shows as "Anywhere on tailscale0 ALLOW". We saw that in our scenario's rules.

Finally, enable UFW if not already:

```
 sudo ufw enable
sudo ufw status verbose
```

Check that the rules make sense. The status might look like:

```
 To                        Action      From
--                       ------      ----
22/tcp (SSH)             ALLOW IN    192.168.0.0/24   (or on tailscale0)
80/tcp (HTTP)            ALLOW IN    Anywhere
443/tcp (HTTPS)          ALLOW IN    Anywhere
8096/tcp (Jellyfin)      ALLOW IN    Anywhere on tailscale0
8096/tcp (Jellyfin)      ALLOW IN    192.168.0.0/24
(and similar v6 if applicable)
Anywhere on tailscale0   ALLOW IN    Anywhere
```

- All other ports are implicitly denied by UFW's default policy. This achieves a locked-down setup where only web traffic and approved private traffic can reach the services.

13. A point to understand deeply: Because Docker can bypass UFW, if we hadn't bound the Nextcloud and Jellyfin to specific interfaces, their ports might be reachable externally even if UFW says deny (since docker inserts its own iptables rules before UFW). To avoid confusion as a master: either use the **binding approach** (as we did for Nextcloud on LAN IP) or use community tools (there are scripts or guides to make UFW and Docker play nicely). In our plan, we assumed binding or at least that UFW rules plus Docker defaults are sufficient – but always test from an external machine to ensure ports are indeed closed/open as expected. A known solution is adding a daemon config for Docker to stop it from altering iptables (then UFW rules truly apply). For completeness, remember this nuance when explaining security.

**Set Up Tailscale for Remote Access:** Now install Tailscale on the server:

```
curl -fsSL https://tailscale.com/install.sh | sh
sudo tailscale up --accept-routes --accept-dns
```

(The above is a quick install script from Tailscale for Ubuntu. Alternatively, install `tailscale` package from apt if available and run `sudo tailscale up`.)

When you run `tailscale up` the first time, it will prompt a URL – visit that in your browser to authenticate the server into your Tailscale network (you'll log in with your Google/Microsoft/other account that you use for Tailscale). Once done, the server is connected. It gets an IP like `100.x.y.z` and a Tailscale DNS name (something like `servername.yourtailnet.ts.net`). You can run `tailscale ip -4` to see the IP.

Install Tailscale on your client machine (laptop/phone) as well and join the same tailnet. Now you can ping the server's Tailscale IP. Try SSH using the Tailscale IP:

```
ssh user@100.x.y.z
```

14. It should work (if you allowed SSH on tailscale0 as above). This is a secure way to SSH without opening port 22 to the entire internet.

    For **Jellyfin access**: On your client, you can now open a browser to `http://<server-tailscale-IP>:8096` or using the MagicDNS name (if you enabled it, e.g. `servername.tailnet.ts.net:8096`). You should see the Jellyfin interface (likely a login or setup screen if first run). The connection is encrypted at the network layer (WireGuard) by Tailscale, even though Jellyfin itself is over HTTP. If you prefer, you could also secure Jellyfin with HTTPS using a reverse proxy or Jellyfin's own settings, but it's typically not necessary when using a VPN like this.

    If you want to go a step further: Tailscale has a feature called **"Tailscale Funnel"** which can expose a service to the public internet via your Tailscale node, with automatic HTTPS. In April 2025, one user used Tailscale's HTTPS support to expose Nextcloud without opening ports by mapping port 443 to a Tailscale funnel. We won't detail that here since we set up our own domain and Nginx, but it's good to know as an alternative approach in case you cannot port-forward or prefer not to run your own Nginx.

15. **Testing and Verification:** At this point, do a full test of the system:

    - **Nextcloud:** Browse to `https://yourcloud.example.com`. Log in, upload a file, create a folder, etc. Verify that the site is using HTTPS (certificate valid), and no errors about trusted domains. Check Nextcloud's *Settings* > *Overview* to ensure all checks pass (no warnings about missing indexes, or memory caches – those are beyond our scope, but for mastery you might mention enabling Redis cache in the future for better performance).

    - **Jellyfin:** From a device on Tailscale or LAN, go to `http://<server>:8096`. The first time, Jellyfin will prompt to create an admin user account and setup libraries. Go ahead and add a media library pointing to `/media/movies` (which corresponds to your `/mnt/data/media/movies` on host). After scanning, try playing a video or music file from a client device. It should stream. If you try to access Jellyfin from a device not on VPN or LAN, it should timeout – which is what we expect (since it's not exposed).

    - **Security:** Use an external port scanner or simply try to reach your IP on ports other than 80/443 (they should be filtered). For example, `nmap -p 1-10000 yourserver` from an external host should show only 22 (if you allowed it globally) and 80/443 open. Ideally, 22 shouldn't be open publicly if using Tailscale-only. Ensure UFW logs or status confirm that.

    - **Resource Check:** Run `docker ps` to list containers – you should see `nextcloud-app`, `nextcloud-db`, and `jellyfin` all "Up". You can run `docker logs nextcloud-app` or `docker logs jellyfin` to see if any errors. Check `sudo systemctl status nginx` to ensure Nginx is active, and maybe `sudo tail -f /var/log/nginx/access.log` to see access entries.

16. **Advanced Maintenance (for Mastery):** Finally, let's discuss a few maintenance and advanced points (these often come up in interviews to gauge deeper understanding):

    ○ *Updates:* How do you update your services? Since we used Docker, updating is as easy as pulling new images and restarting containers. For Nextcloud, you should follow the official upgrade path (sometimes it's recommended to use the built-in updater inside Nextcloud for major version upgrades). But you can generally do `docker compose pull && docker compose up -d` to fetch the latest images. Always back up data volumes before major upgrades.

    ○ *Backups:* A true master has a backup strategy. Our data resides in volumes: Nextcloud files (in `/mnt/data/nextcloud` and the database in `/mnt/data/db`), and Jellyfin config (in `/mnt/data/jellyfin`) plus your media on `/mnt/data/media`. You should back up the Nextcloud database and the files regularly (the Nextcloud interface also provides server-side encryption and external storage features – beyond scope). You might use `mysqldump` to backup the MariaDB, or stop the container and copy the files. Having snapshots or rsync backups to another drive or cloud is wise.

    ○ *Security Hardening:* We did the basics (UFW, HTTPS). Additional steps could include setting up Fail2Ban to ban IPs that hammer the Nextcloud login, using Nextcloud's brute-force protection (enabled by default), and perhaps configuring Jellyfin with a strong password (Jellyfin has user accounts, ensure the default admin is protected). We might also want to configure **APCu** or Redis as memory cache for Nextcloud if expecting heavy usage – this improves performance (would require adding a Redis container and adjusting config).

    ○ *Monitoring:* Use tools like `docker stats` to monitor container resource usage, or set up something like Netdata or Prometheus for system monitoring. Being able to discuss how you'd scale this setup is also a plus: e.g. for more users, you might move to an external database, or use multiple Nextcloud app servers behind a proxy, etc., but for a personal setup that's overkill.

    ○ *Understanding Docker Networking:* As a deep-dive, note that by default Docker Compose put Nextcloud and DB in a common user-defined network so they could communicate by service name. We published Nextcloud's port to host. If we had not specified the host IP, Docker would listen on 0.0.0.0 (all interfaces). This is why we carefully controlled UFW. Knowing how to inspect Docker's iptables rules (`sudo iptables -L -n`) could impress an interviewer. You can mention that Docker creates a `DOCKER` chain in iptables that bypasses UFW's chains by default. The fix we discussed: set Docker's `"iptables": false` if you want UFW to fully control traffic – but then you must manually allow inter-container traffic or other rules. Many people use community scripts (like `ufw-docker` wrapper) to manage this.

- *Why use reverse proxy vs direct:* We should be able to explain why we bothered with Nginx at all. The key reasons: to handle multiple services on one host (Nginx can host Nextcloud on 443, maybe something else on another subdomain, etc.), and to terminate SSL (Nextcloud's built-in Apache *could* do SSL too, but managing Let's Encrypt renewal across Docker might be trickier; we chose a host-based approach for simplicity and transparency). Also, tools like Nginx give more control (rate limiting, additional web application firewall rules, etc., if needed).

- *Alternate approaches:* A master knows there are many ways to do this. For instance, one could use Docker images from linuxserver (community) that bundle Nginx and Certbot via a companion container (like Nginx Proxy Manager or Caddy server to automate certs). We took a more manual educational route. One could also install Nextcloud directly on the host (LAMP stack) without Docker – but then isolating Jellyfin and dependencies might be harder. Docker gave us an easy sandbox for each app.

- *Scalability & Performance:* For a home server, our setup on a single machine is fine. In an enterprise, you'd separate concerns (database on a different server, use external storage, etc.). If an interviewer pushes, mention that Nextcloud can use **Redis** for file locking and memcache (improves performance), and maybe **object storage** (S3-compatible) for storing files instead of local disk, and **CDN** integration for serving files if at large scale. Jellyfin can leverage hardware acceleration for transcoding videos (requires giving the container access to GPU, e.g., using `--device` flags for VAAPI/NVENC).

- *Logging and Troubleshooting:* Know where logs are. Nextcloud logs (inside the container, or via the Nextcloud web UI log viewer). Nginx logs in `/var/log/nginx`. Jellyfin logs in its config directory. If something isn't working, these are your first stops. For example, if after setting up proxy you got a blank page, Nginx error log might show what's wrong (maybe proxy_pass couldn't connect because container wasn't listening on expected interface).

By covering the above, you can confidently **explain every component of the setup** and answer questions like:

- *“Why use Docker and not a normal install?”* – You can discuss isolation, easier dependency management, and consistency (plus quick recovery – redeploy container quickly if needed).

- *“How does Jellyfin differ from something like Plex?”* – Jellyfin is open source and doesn’t require any central account – all data stays with you, aligning with the privacy ethos of self-hosting. Technical differences might not be asked unless the role is specific, but you can mention Jellyfin is a fork of Emby and uses similar DLNA/streaming tech.

- *“How do clients securely connect to Jellyfin when it’s not public?”* – Through Tailscale VPN which encrypts traffic (WireGuard) and acts as if the client is in the server’s local network. This is a Zero Trust approach – only devices you authorize can even see the service.

- *“What happens if your certificate expires?”* – Certbot installed a cron job, so it will auto-renew every ~60 days. We can run `sudo certbot renew --dry-run` to test renewal. As a master, you’d monitor your emails for any renewal issues.

- *“How would you add another service to this setup?”* – For example, you could add an application like **Cockpit** (web admin) or **Grafana** – you’d likely run them in Docker or directly, and use Nginx to give them a subdomain or port. In fact, the user’s original context showed they had **Cockpit** (on port 9090) and they disabled it (perhaps to free resources). If you mention that, say Cockpit is a web GUI for server management – they turned it off perhaps for security once comfortable with command line. It’s good to show awareness of such utilities.

## Conclusion

Congratulations on setting up a robust self-hosted environment! We implemented a personal cloud with Nextcloud for file syncing and Jellyfin for media streaming, using Docker to containerize applications and Nginx + Let’s Encrypt to secure the web interface. We also locked down the server with a firewall and enabled Tailscale VPN for private access to non-public services. Along the way, we delved into the *“why”* behind each choice – so you not only know the commands, but also the concepts to **explain flawlessly how it all works**. By going from basic principles to advanced tweaks, you’ve gone from *noob* to *master* on this topic. This setup can now be re-deployed by you without needing to look up instructions, and you can adapt it to future needs (for example, hosting additional apps, scaling up resources, etc.).

Always keep learning and tinkering – the world of self-hosting and DevOps is vast. With this foundation, you can confidently tackle interview questions about web servers, containers, networking, and security. Happy self-hosting!

**Sources:** To reinforce the concepts, here are some references:

- Nextcloud’s open-source nature and similarity to Dropbox/Drive, and its powerful self-hosted collaboration features.
- Jellyfin’s mission of putting you in control of your media.
- Docker container fundamentals – sharing the host OS kernel for efficiency.
- Community solution for Nextcloud behind proxy: setting trusted_proxies and overwriteprotocol.
- Docker and UFW interaction – Docker’s iptables bypass can surprise firewall configurations.
- Tailscale’s ability to connect devices without manual firewall rules (NAT traversal).