Ease your Docker workflow: Save bandwidth, work efficiently.

Why Do This?

We often rely on solutions like Google Drive, but they have limitations:

1. **Time and Cost Efficiency:** Building our own file server reduces reliance on third-party vendors, saving on costs.
2. **Vendor Independence:** Avoid being locked into a particular service provider.
3. **Maximizing Resources:** Utilize our own server space to store files and container images, optimizing deployment workflows.

Technologies & Components Used

- **Python 3.12 & Starlette (with JWT)**: For the backend API handling.
- **Nginx with Lua**: To handle HTTP requests efficiently.
- **Bash**: For automating tasks and server configurations.
- **Docker**: To containerize the application and ensure scalability.
- **Ubuntu 24.04**: The operating system running the server.

Course Overview

In this course, we'll guide you through building a scalable, production-worthy file server. By the end of the training, you will be equipped to:

1. Upload and download files via an API.
2. Configure Nginx to ensure fast and secure file delivery.
3. Use Docker effectively to streamline deployments.

You'll have access to the full source code, and you're encouraged to explore further at your own pace. If you have any questions, feel free to contact me via **ichux** on the PythonNigeria Slack Channel, LinkedIn, or **@zuoike** on X.

Assumptions

1. Your PC has sufficient RAM to handle the workload.
2. You are running a Linux kernel (may work on Mac, but not tested).
3. Windows users will need alternative methods to test the setup.

Course Advantages

1. **Secure File Downloads:** You can't download a file unless you:

   - Know the file path on the server.
   - Have a valid JWT token (which expires after a set time).

   Even if a file is still on the server, you cannot access it without the token, enhancing security.

2. **Content-Disposition Header:** Handled by Nginx, ensuring correct file delivery to the client.

3. **File Size Limitation:**

- The API has an upload size limit, but you can bypass this by using SCP for file transfers.

4. **Efficient Uploads with Streaming:**
    - The API uses streaming for uploads, ensuring smooth transfer even for large files.
    - Proper permissions are set to facilitate easy file retrieval.
5. **Fast Resumable Downloads:**
    - Nginx efficiently handles resumable downloads for large files.
6. **Optimized Docker Setup:**
    - Multi-stage builds minimize the size of the final image.
    - Path settings ensure efficient reuse of images, reducing time and storage space.
7. **Database Migrations:**
    - Alembic is used for database migrations, ensuring smooth schema changes.
8. **Separation of Concerns:**
    - A dedicated container for managing the web app (e.g., handling migrations) keeps the system organized.
9. **User Consistency:**
    - The system user matches the Docker user, preventing permission-related issues.

Upload & Download Methods

1. **File Uploads:**
    - You can upload files using either **SCP** or an API that requires a username and password.
2. **File Downloads:**
    - JWT is used for token-based authentication during file downloads.

   Note: SCP is outside the scope of this course, but you're encouraged to research it on your own after the course.

Potential Use Cases for the Source Code

1. **Backup Solution:** Deploy as-is and use it as a file backup system for all your media.
2. **Media Streaming:** Download and stream files at your convenience. You can even offer file storage services to others for a fee.
3. **CI/CD Integration:** Use the source code to automatically push files to your file server through a CI/CD pipeline.
4. **Time-Saving Workflow:** Leverage the source code to save significant time in your workflow, especially when dealing with images and containerized applications.

Optimizations Implemented

- **System User Inside Docker:** Ensures proper file permission management on Ubuntu.
- **Reusing Anchors and Aliases:** Reduces redundancy in the Dockerfile for better maintainability.
- **Multi-Stage Builds:** Optimizes the Docker image, ensuring faster builds and smaller image sizes.

- **Minimal Image Usage:** Ensures that only necessary components are included in the final image.
- **Compression:** Compresses images to save space and bandwidth during deployment.