Files

Doku.md

Doku.md 9.62 KiB

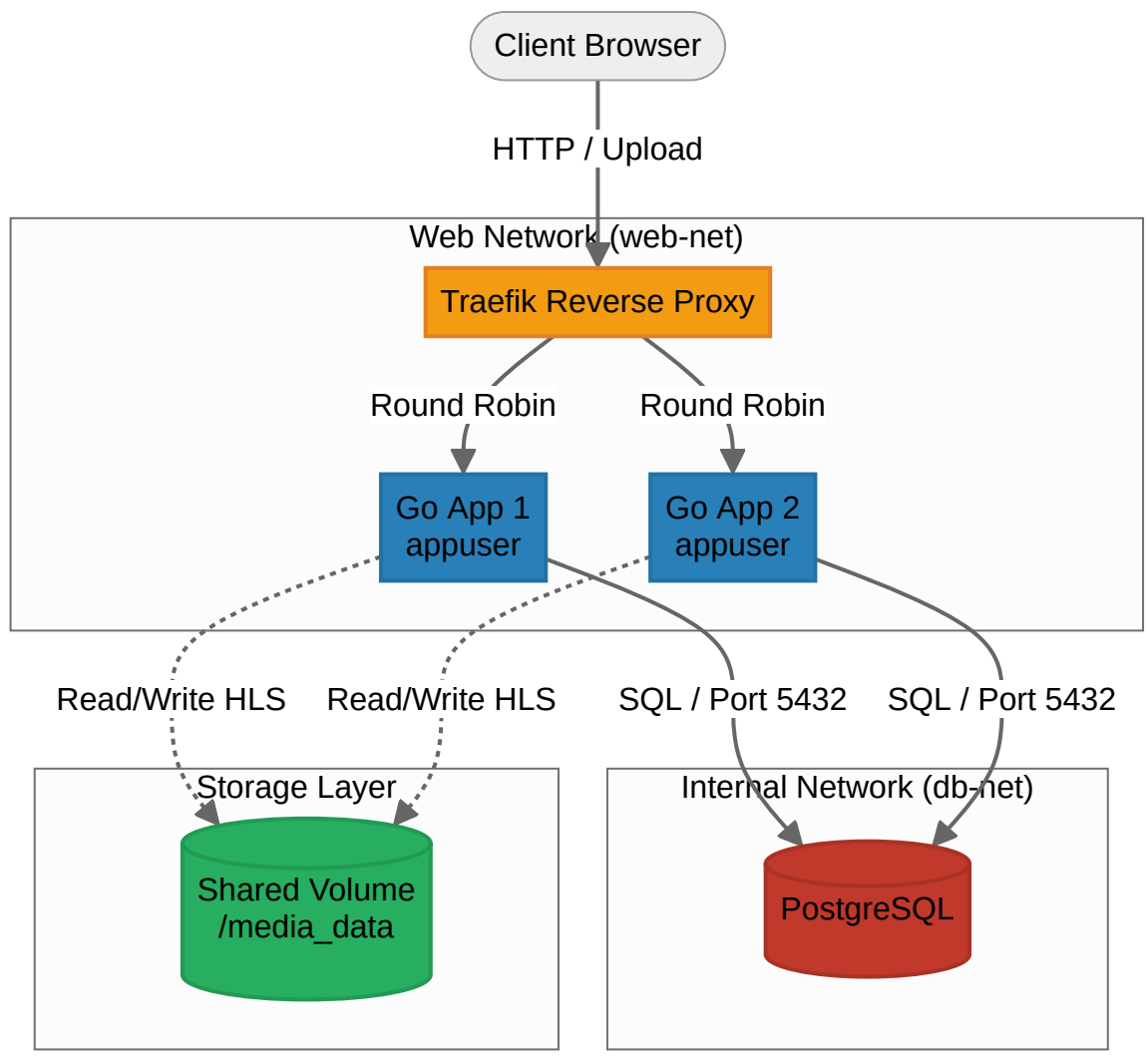
MeTube – Distributed Video Transcoding Platform

Projektübersicht

MeTube ist eine skalierbare Web-Applikation zur asynchronen Verarbeitung von Video-Transcoding-Aufgaben. Das System wandelt hochgeladene MP4-Videos in das **HLS-Format (HTTP Live Streaming)** um.

Der Fokus liegt auf der horizontalen Skalierbarkeit: Mehrere Instanzen des Dienstes teilen sich die Last. Fällt eine Instanz aus, übernimmt eine andere automatisch die ausstehenden Aufgaben (**Self-Healing & Failover**).

Visualisierung



System-Architektur

Das System besteht aus einem Cluster von Diensten, die über Docker Compose koordiniert werden:

- **Load Balancer (Traefik):** Verteilt eingehende Anfragen per Round-Robin auf die verfügbaren Go-Instanzen.
- **Backend (Go-Services):** Identische Instanzen (`app-1` , `app-2`), die zustandslos (stateless) arbeiten.
- **Datenbank (PostgreSQL):** Zentrale "Single Source of Truth" für Benutzerdaten, Job-Status und das Locking-Verfahren.
- **Shared Storage:** Ein Docker-Volume, das alle Instanzen nutzen, um auf Rohdaten und die generierten HLS-Segmente zuzugreifen.

Web-Architektur

Das Projekt besitzt eine klassische SPA mit einer index.html und einem app.js.
Es besitzt CSR Elemente wie die /my-videos API, bei der es eine JSON Dokument bekommt und dieses dann rendert.
Das Go-Backend fundiert hier als stateless REST-API.

Sicherheits- & Auth-Konzept

Wir nutzen ein **Hybrid-JWT-Verfahren**, um Sicherheit und Performance im Cluster zu garantieren:

1. **Access Token (Short-lived):** Wird im Frontend (JavaScript-Variable) gehalten. Erlaubt den Zugriff auf geschützte Routen wie `/upload` oder `/my-videos`.
2. **Refresh Token (Long-lived):** Gespeichert in einem **HttpOnly Cookie**. Dies schützt vor XSS-Angriffen, da JavaScript den Token nicht auslesen kann.
3. **Silent Refresh:** Das Frontend erneuert den Access Token automatisch über den `/refresh` Endpunkt, ohne dass der Nutzer seine Daten erneut eingeben muss.
4. **User-Job-Mapping:** Jeder Transcoding-Job ist in der Datenbank fest mit einer `user_id` verknüpft. Nutzer sehen so ihre eigenen Videos.

Installation & Start

Voraussetzungen

- Docker & Docker Compose installiert.
- Eine `.env` Datei im Hauptverzeichnis mit folgendem Inhalt:

```
DB_PASSWORD=dein_db_passwort
ACCESS_TOKEN_SECRET=dein_geheimer_access_key
REFRESH_TOKEN_SECRET=dein_geheimer_refresh_key
```

Startbefehl

```
docker compose up -d --build
```

Das System ist anschließend unter `http://localhost` erreichbar. Das Traefik-Dashboard findest du unter `http://localhost/dashboard`. Desweiteren ist noch eine Monitor Webseite verfügbar, bei der man den aktuellen Stand der Transcodings sehen kann. Diese ist unter `http:localhost/monitor` erreichbar.

API-Endpunkte

Methode	Pfad	Authentifizierung	Beschreibung
`POST`	`/register`	Öffentlich	Erstellt einen neuen Benutzer.
`POST`	`/login`	Öffentlich	Gibt Access-Token aus & setzt Refresh-Cookie.
`GET`	`/stats`	Öffentlich	Zeigt den aktuellen Status des Clusters.
`POST`	`/refresh`	Refresh-Cookie	Erzeugt neuen Access-Token (Silent Refresh).
`GET`	`/my-videos`	JWT	Listet alle Videos des aktuellen Users.
`POST`	`/upload`	JWT	Startet das Transcoding eines Videos.
`GET`	`/status/id`	JWT	Gibt den Status eines spezifischen Jobs aus

Resilienz & Failover

Das System ist darauf ausgelegt, den Absturz einer Instanz während des Transcodings zu überleben:

- **State Tracking:** Jede Instanz markiert einen Job in der DB als `PROCESSING`.
- **Timeout Recovery:** Ein Hintergrund-Prozess prüft regelmäßig auf "verwaiste" Jobs. Wenn eine Instanz abstürzt, wird der Job zurückgesetzt und von einer gesunden Instanz neu gestartet.
- **Stateless Scaling:** Da die Authentifizierung über JWT erfolgt, ist es egal, welche Instanz die Anfrage bearbeitet – solange das Secret identisch ist.

Loadtesting

Der Hauptbottleneck ist ganz klar das Transcoding, da dieses relativ viel CPU Power braucht und jeweils nur ein Dienst an einem File arbeiten kann. Allerdings können natürlich mehrere Container gestartet werden, um viele Files gleichzeitig zu verarbeiten.

Ich habe einen Loadtest von meinen Computer zu Hause (Spezifikation folgt) durchgeführt auf die normale http://localhost Seite, um den Loadbalancer auszureizen. Für den Loadtest habe ich hey verwendet:

Spezifikationen:

```
OS: CachyOS x86_64
Host: X570 Phantom Gaming 4S
Kernel: Linux 6.19.10-1-cachyos
CPU: AMD Ryzen 9 5900X (24) @ 4.95 GHz
GPU: NVIDIA GeForce RTX 3080 [Discrete]
Memory: 32GB 4400MHz
```

Loadtest

```
hey -z 30s http://localhost

Summary:
  Total:        30.0012 secs
  Slowest:      0.0274 secs
  Fastest:      0.0002 secs
  Average:      0.0015 secs
  Requests/sec: 53554.5713

  Total data:   3693805599 bytes
  Size/request: 3693 bytes

Response time histogram:
  0.000 [1]     |
  0.003 [997765]        |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.006 [2120]  |
  0.008 [3]     |
  0.011 [12]    |
  0.014 [3]     |
  0.016 [28]    |
  0.019 [32]    |
  0.022 [0]     |
  0.025 [0]     |
  0.027 [36]    |


Latency distribution:
  10%% in 0.0005 secs
  25%% in 0.0006 secs
  50%% in 0.0009 secs
  75%% in 0.0011 secs
  90%% in 0.0014 secs
  95%% in 0.0017 secs
  99%% in 0.0023 secs
```

```
Details (average, fastest, slowest):
  DNS+dialup:   0.0000 secs, 0.0000 secs, 0.0051 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0008 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0029 secs
  resp wait:    0.0014 secs, 0.0002 secs, 0.0256 secs
  resp read:    0.0000 secs, 0.0000 secs, 0.0020 secs

Status code distribution:
  [200] 1000000 responses
```

Gegen Endpoint mit DB Abfrage:

```
hey -z 30s -H "Authorization: Bearer [TOKEN]" http://localhost/my-videos
Summary:
  Total:        30.0209 secs
  Slowest:      0.0781 secs
  Fastest:      0.0003 secs
  Average:      0.0090 secs
  Requests/sec: 5567.0143

  Total data:   147740268 bytes
  Size/request: 884 bytes

Response time histogram:
  0.000 [1]     |
  0.008 [118444]        |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.016 [1843]  |■
  0.024 [16684] |■■■■■■
  0.031 [21725] |■■■■■■■
  0.039 [7108]  |■■
  0.047 [1138]  |
  0.055 [164]   |
  0.063 [15]    |
  0.070 [4]     |
  0.078 [1]     |


Latency distribution:
  10%% in 0.0009 secs
  25%% in 0.0013 secs
  50%% in 0.0023 secs
  75%% in 0.0201 secs
  90%% in 0.0278 secs
  95%% in 0.0315 secs
  99%% in 0.0384 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0000 secs, 0.0000 secs, 0.0041 secs
  DNS-lookup:   0.0000 secs, 0.0000 secs, 0.0030 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0026 secs
  resp wait:    0.0089 secs, 0.0003 secs, 0.0781 secs
  resp read:    0.0000 secs, 0.0000 secs, 0.0020 secs

Status code distribution:
  [200] 167127 responses
```

Probleme und Hürden

Traefik Konfiguration

Hier hatte ich Mühe wegen der Konfiguration des Traefik Containers, da es mir von der AI, sowie von der Traefik Webseite immer empfohlen hat den Docker Socket mitzugeben. Wie wir aber in der Vorlesung gelernt haben ist dies eine sehr schlechte Idee und ich wollte unbedingt umgehen. Hier hat dann die Vorlesung über Loadbalancing geholfen und ich konnte die Konfiguration dementsprechend anpassen.

Traefik Healthchecks

Zuerst habe ich die Application Container mittels `deploy` vermehrt. Dies hat dann zu DNS Problemen geführt, dass die Routes zuerst gar nicht mehr funktioniert haben. Dies habe ich dann angepasst, allerdings hat mir der Healthcheck dann immer angezeigt, dass die Container laufen, auch wenn nur einer funktioniert hat. Deswegen habe ich diese dann in unterschiedliche Instanzen getrennt.

Go

Dies war das erste Projekt mit Go, weshalb ich mir hier auch stark Hilfe von AI geholt habe. Ich bin erstaunt, wie elegant Go ist und wie "einfach" sich die Syntax liest. Auch wenn ich momentan nur fähig bin Anpassungen zu machen, ohne direkt wieder die Docs zu öffnen, finde ich lässt sich klar lesen können was gerade in dem jeweiligen Befehl gemacht wird.

Gebrauch von AI

AI wurde stark für das Coding verwendet. Der grösste Teil des Codes wurde mit AI generiert. Allerdings sind die Finetuning und Gedanken von mir eingeflossen.
Auch die Wahl der Architektur wurde von mir vorgegeben. Das Docker compose wurde dann gegen Ende nahezu nur von mir geschrieben.