ChatGPT

1. **Docker Engine Installation & Swarm Initialisierung**
   Beschreibung: Auf dem Zielserver muss Docker installiert und anschließend der Swarm-Modus aktiviert werden. Dies richtet einen Ein-Knoten-Docker-Swarm ein, der als Grundlage für das Deployment dient.
   Ziel: Einen Docker-Swarm (Single-Node) als Umgebung für den Stack aufsetzen [1].
   Verantwortlich: DevOps
   Durchführung: Führen Sie zunächst die Installation von Docker durch (falls noch nicht vorhanden) und initialisieren Sie dann den Swarm:

   ```
   # Docker installieren (für Debian/Ubuntu)
   sudo apt-get update && sudo apt-get install -y docker.io
   sudo systemctl enable --now docker

   # Aktuellen Nutzer zur Docker-Gruppe hinzufügen (optional, für
   berechtigungsloses Ausführen)
   sudo usermod -aG docker $USER

   # Docker Swarm initialisieren (Single-Node Swarm)
   docker swarm init --advertise-addr <SERVER-IP>  # <SERVER-IP> durch die
   IP des Servers ersetzen
   ```

   Hinweis: `docker swarm init` schaltet den aktuellen Knoten in den Swarm-Modus und macht ihn zum Manager des neuen Swarms [1]. Bei Erfolg erscheint `"Swarm initialized"` in der Ausgabe.

2. **GitHub Container Registry (GHCR) vorbereiten**
   Beschreibung: Einrichten der Container Registry auf GitHub, um die gebauten Images dort bereitzustellen. Es wird ein Personal Access Token (PAT) mit Package-Rechten benötigt, um sich bei der GHCR anzumelden. Anschließend wird der Login auf Entwicklungs- **und** Server-Seite durchgeführt.
   Ziel: Zugriff auf die GitHub Container Registry sicherstellen und Container-Images in GHCR veröffentlichen können [2] [3].
   Verantwortlich: DevOps
   Durchführung: Erstellen Sie in GitHub einen Personal Access Token (classic) mit Scope `write:packages` (und `read:packages` für Pull) im GitHub-Konto des Projekt-Owners. Melden Sie sich dann lokal **und auf dem Server** bei `ghcr.io` an:

   ```
   # Personal Access Token als Umgebungsvariable (Beispiel)
   export CR_PAT="<GHCR_Personal_Access_Token>"

   # Bei GitHub Container Registry anmelden
   echo $CR_PAT | docker login ghcr.io -u <GitHub-Username> --password-
   stdin
   ```

   Bei erfolgreichem Login erscheint *"Login Succeeded"*. Führen Sie diesen Login-Befehl auf dem Server (Swarm-Manager) aus, damit dieser später private Images ziehen kann. Standardmäßig

sind neu gepushte GHCR-Images privat [4]. Optional können Sie die Sichtbarkeit der Images in den GitHub Packages-Einstellungen des Repos auf **public** stellen, um Pulls ohne Authentifizierung zu ermöglichen. Andernfalls muss `docker login` auf jedem Swarm-Knoten durchgeführt werden oder beim Deployment das Flag `--with-registry-auth` verwendet werden (siehe Task 9).

3. **Dockerfiles für Services erstellen**
   Beschreibung: Für jeden Service/Microservice im Repository wird ein Dockerfile angelegt oder überprüft. Die Dockerfiles sollten die Applikation entsprechend der Entwicklungsumgebung paketieren (Basis-Image wählen, Abhängigkeiten installieren, Anwendung starten). Dabei sind Ports und Pfade gemäß den Anforderungen des Dienstes zu konfigurieren.
   Ziel: Jeder Service des Projekts besitzt ein funktionierendes Dockerfile, sodass ein Container-Image gebaut werden kann. Dadurch wird sichergestellt, dass alle Komponenten containerisiert sind und im Swarm laufen können.
   Verantwortlich: Entwickler
   Durchführung: Legen Sie im Wurzelverzeichnis jedes Service (laut Repository-Struktur) eine Datei `Dockerfile` an. Im Folgenden ein Beispiel für einen Python-basierten Backend-Service (bei Verwendung von z.B. Flask/FastAPI):

```
# Beispiel-Dockerfile für Backend-Service (Python)
FROM python:3.10-slim

# Arbeitsverzeichnis setzen
WORKDIR /app

# Abhängigkeiten kopieren und installieren
COPY requirements.txt .
RUN pip install -r requirements.txt

# Anwendungscode kopieren
COPY . .

# Port (intern) und Startbefehl definieren
EXPOSE 8000
CMD ["python", "app.py"]
```

   Hinweis: Passen Sie das Dockerfile an die jeweilige Technologie an (z.B. Node.js, Java, etc.). Für einen Frontend-Service könnte z.B. ein Nginx als Base-Image genutzt werden, der gebaute statische Dateien ausliefert. Wichtig ist, dass jedes Dockerfile am Ende einen geeigneten Entrypoint bzw. CMD zum Start der Applikation enthält. Stellen Sie außerdem sicher, dass alle benötigten Umgebungsvariablen von der Anwendung ausgelesen werden (z.B. DB-Verbindungsdaten), damit diese später im Compose-File gesetzt werden können.

4. **Container-Images bauen und nach GHCR pushen**
   Beschreibung: Sobald die Dockerfiles vorhanden sind, müssen die Images für alle Services gebaut und in die GitHub Container Registry hochgeladen werden. Die Images sollen mit dem Namespace des GitHub-Accounts (oder der Org) und einem geeigneten Tag (z.B. `latest` oder Versionsnummer) versehen werden.
   Ziel: Alle Service-Images sind in GHCR verfügbar (als `ghcr.io/<NAMESPACE>/<IMAGE_NAME>:<TAG>`), sodass der Swarm-Stack sie herunterladen kann.

Verantwortlich: Entwickler
Durchführung: Führen Sie auf der Entwicklungsmaschine (oder dem CI-Server, falls später eingerichtet) für jeden Service die Build- und Push-Befehle aus. Ersetzen Sie `<NAMESPACE>` durch Ihren GitHub-Benutzernamen oder Orga-Namen (z.B. `flissel`), und wählen Sie `<IMAGE_NAME>` passend zum Service. Beispiel für **Backend** und **Frontend** Services:

```
# Backend-Service Image bauen
docker build -t ghcr.io/<NAMESPACE>/unityplatform-backend:1.0.0 ./
backend

# Image in GHCR pushen
docker push ghcr.io/<NAMESPACE>/unityplatform-backend:1.0.0

# Frontend-Service Image bauen
docker build -t ghcr.io/<NAMESPACE>/unityplatform-frontend:1.0.0 ./
frontend

# Image in GHCR pushen
docker push ghcr.io/<NAMESPACE>/unityplatform-frontend:1.0.0
```

Verwenden Sie den aktuellen Tag (z.B. `latest` oder SemVer wie `1.0.0`) konsistent. Nach dem ersten Push überprüfen Sie auf GitHub unter **Packages** des entsprechenden Repositorys oder Benutzerkontos, ob die Images angekommen sind. Denken Sie daran: Das erste Pushen erzeugt das Package automatisch, und es ist zunächst privat [4]. Stellen Sie also sicher, dass der Server Berechtigung hat, das Image zu ziehen (siehe Task 2). Sie können die Images nach dem Push ggf. auf **public** stellen, falls offene Zugänglichkeit gewünscht ist.

5. **Docker-Compose Stack-Datei erstellen (Grundstruktur)**
Beschreibung: Erstellen Sie eine `docker-compose.yml` (Version 3.8 oder höher) als Stack-Definition für Docker Swarm. Legen Sie darin zunächst die allgemeine Struktur, Netzwerke und Volumes fest, die zwischen den Services genutzt werden. Gemäß Architekturdiagramm werden interne und externe Netzwerke benötigt: ein externes Netzwerk für den Traefik-Router und ein internes für die interne Kommunikation (z.B. Backend ↔ Datenbank). Volumes werden für persistente Daten vorgesehen (Datenbank, ggf. Traefik-Zertifikate).
Ziel: Bereitstellen eines Stack-Definitionsfiles mit definierten Netzwerken und Volumes, als Grundlage für das Hinzufügen der Services im nächsten Schritt.
Verantwortlich: DevOps
Durchführung: Legen Sie im Repository (z.B. im Projektwurzelverzeichnis) die Datei `docker-compose.yml` an mit folgendem Grundgerüst:

```
version: '3.8'

networks:
  traefik-public:
    driver: overlay
  internal:
    driver: overlay

volumes:
```

```
  db-data:      # Persistente Datenbank-Daten
  traefik-cert: # Persistente Traefik-Zertifikate (für TLS, optional)
```

Erläuterung:

- **traefik-public**: Dieses Overlay-Netzwerk dient der Kommunikation zwischen Traefik und den publiken Services. (Falls Traefik als separater Stack genutzt würde, könnte dieses Netzwerk als extern deklariert werden. Hier definieren wir es im Stack.)
- **internal**: Overlay-Netz für die interne Kommunikation (z.B. Backend mit DB). Services, die nicht vom Traefik aus erreichbar sein sollen (z.B. die DB), kommen nur in dieses Netz.
- **db-data Volume**: Behält die Datenbankdaten (z.B. PostgreSQL) persistent auf dem Host.
- **traefik-cert Volume**: Speichert Zertifikate von Traefik (z.B. Let's Encrypt ACME-Datei), damit diese bei Container-Neustart erhalten bleiben. (Wenn anfänglich kein TLS/Let's Encrypt genutzt wird, kann dies trotzdem vorbereitet werden.)

6. **Traefik-Service zum Stack hinzufügen**
   Beschreibung: Den Reverse Proxy Traefik als Service in die Compose-Datei integrieren. Dieser Service lauscht auf Ports 80/443 und leitet eingehende Requests anhand definierter Regeln an die internen Services weiter. Konfigurieren Sie Traefik so, dass es das Docker-Swarm-API ausliest (docker provider) und definieren Sie benötigte Ports, Volumes und Netzwerke. Der Traefik-Dashboard-Zugang kann optional geschützt oder deaktiviert werden – hier liegt Fokus auf dem Routing.
   Ziel: Traefik läuft als Swarm-Service und übernimmt das Routing für alle freizugebenden Dienste (laut Architekturdiagramm der zentrale Einstieg).
   Verantwortlich: DevOps
   Durchführung: Fügen Sie in `docker-compose.yml` den **Traefik** Service unter `services:` hinzu:

```
services:
  traefik:
    image: traefik:2.9              # Verwenden der offiziellen
Traefik-Image-Version
    command:
      - --providers.docker.swarmMode=true    # Swarm-Modus aktivieren,
liest Services/Labels aus Swarm 5
      - --providers.docker.network=traefik-public
      - --entryPoints.web.address=:80        # HTTP EntryPoint
      - --entryPoints.websecure.address=:443 # HTTPS EntryPoint
      - --api=true --api.dashboard=true      # (Optional) Dashboard
aktivieren
      - --log.level=INFO
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro   # Docker-Socket
für Traefik (ReadOnly)
      - traefik-cert:/certificates                     # Volume für
Zertifikate (z.B. ACME)
    networks:
```

```
      - traefik-public
    deploy:
      placement:
        constraints:
          - node.role == manager   # Traefik nur auf Manager-Knoten
ausführen (erhält Docker-Events)
      labels:
        - traefik.enable=true
        - traefik.http.routers.ping.rule=Path(`/ping`)
        - traefik.http.routers.ping.service=ping@internal
        - traefik.http.services.ping.loadbalancer.server.port=8082
```

Erläuterung:

- Wir setzen Traefik auf, damit es mittels Docker-Provider die Service-Labels ausliest. Das Label-Format ist so gewählt, dass in Swarm-Mode die Labels am **Service** hängen (unter `deploy`) [5].
- Die Ports 80 und 443 des Host-Servers werden an Traefik weitergeleitet (hier ohne TLS-Termination konfiguriert – TLS kann via Traefik LetsEncrypt später ergänzt werden).
- Das Docker-Socket wird als Read-Only Volume gemountet, damit Traefik die Container/Service-Informationen und Labels aus dem Swarm lesen kann.
- Das `traefik-cert` Volume ist gemountet auf `/certificates` (wird von Traefik für ACME-Zertifikatsspeicherung genutzt, optional).
- Die Beispiel-Labels definieren einen einfachen ping-Check Router (Aufruf `/<host>/ping` liefert intern von Traefik einen 200-OK, dient als liveliness test). Für eigene Services werden in den nächsten Schritten weitere Router-Labels definiert.
- Die `placement` Constraint stellt sicher, dass Traefik auf dem Manager läuft. In einem Multi-Node-Szenario ist das üblich, da nur Manager alle Events sehen. (Für Single-Node hat es keine Auswirkung, da dieser Manager ist.)

7. **Datenbank-Service zum Stack hinzufügen**
   Beschreibung: Integration des Datenbank-Containers gemäß Architektur (z.B. PostgreSQL oder MongoDB). Dieser Service soll im Compose-Stack konfiguriert werden mit einem persistenten Volume für die Daten und den nötigen Umgebungsvariablen (z.B. Benutzer, Passwort, Datenbankname). Der DB-Container wird **nur** im internen Netzwerk betrieben, ohne externe Exposition.
   Ziel: Die persistente Datenbank läuft als Service im Swarm und ist für die benötigten anderen Services erreichbar, aber nicht öffentlich zugänglich.
   Verantwortlich: DevOps (in Absprache mit Entwickler bezüglich DB-Einstellungen)
   Durchführung: Fügen Sie z.B. einen PostgreSQL-Service der Compose-Datei hinzu:

```
  db:
    image: postgres:15-alpine           # Offizielles PostgreSQL Image
    volumes:
      - db-data:/var/lib/postgresql/data
    environment:
      - POSTGRES_USER=unityplatform     # Datenbank-Benutzername
      - POSTGRES_PASSWORD=secretpass    # Passwort (starkes Passwort
setzen!)
      - POSTGRES_DB=unity               # Initial anzulegende Datenbank
```

```
    networks:
      - internal
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
```

Erläuterung: In diesem Beispiel wird eine PostgreSQL-DB mit dem Volume `db-data` verknüpft, sodass Daten auf dem Host untergebracht werden. Die Env-Variablen legen Benutzer, Passwort und DB-Namen fest – stimmen Sie diese Werte mit den Verbindungs-Einstellungen in der Anwendung ab. Der Service `db` befindet sich **nur** im `internal`-Netz und **ohne** Ports-Mapping, d.h. er ist von außen nicht direkt erreichbar (was aus Sicherheitsgründen gewollt ist). Die Anwendung wird über das interne Netzwerk auf diesen Service zugreifen (Hostname innerhalb des Swarm-Netzwerks = "`db`").

8. **Backend- und Frontend-Services zum Stack hinzufügen**
   Beschreibung: Integration der Anwendungs-Services (z.B. Backend-API und Frontend) in die Compose-Datei. Diese Services nutzen die vorher erstellten Container-Images aus GHCR. Setzen Sie die notwendigen Environment-Variablen (z.B. DB-Verbindungsstrings, API-Keys) und konfigurieren Sie die Netzwerke so, dass das Backend mit der DB kommunizieren kann (internes Netz) und Traefik externe Anfragen an die richtigen Services weiterleitet (Traefik-Netz und Labels). Der Frontend-Service (falls vorhanden, z.B. ein Webserver mit statischen Dateien) wird so konfiguriert, dass Traefik ihn erreichbar macht.
   Ziel: Die Anwendungskomponenten laufen als Dienste im Swarm und sind gemäß Architektur miteinander verbunden – das Backend spricht mit der DB, und Traefik kann Anfragen an Backend/Frontend routen.
   Verantwortlich: DevOps (in Absprache mit Entwicklern für korrekte Env-Einstellungen)
   Durchführung: Fügen Sie der Compose-Datei z.B. folgende Services hinzu:

```
  backend:
    image: ghcr.io/<NAMESPACE>/unityplatform-backend:1.0.0
    environment:
      - DB_HOST=db
      - DB_PORT=5432
      - DB_NAME=unity
      - DB_USER=unityplatform
      - DB_PASS=secretpass
      # ... weitere ENV (z.B. API Keys, Konfigs laut Bedarf)
    networks:
      - internal
      - traefik-public
    depends_on:
      - db
    deploy:
      replicas: 1
      labels:
        - traefik.enable=true
        - traefik.http.routers.backend.rule=Host(`api.meine-domain.de`)
        - traefik.http.routers.backend.entrypoints=websecure
        - traefik.http.routers.backend.tls.certresolver=le
```

```
# Optional: Let's Encrypt nutzen
        - traefik.http.services.backend.loadbalancer.server.port=8000

  frontend:
    image: ghcr.io/<NAMESPACE>/unityplatform-frontend:1.0.0
    networks:
      - traefik-public
    deploy:
      replicas: 1
      labels:
        - traefik.enable=true
        - traefik.http.routers.front.rule=Host(`app.meine-domain.de`)
        - traefik.http.routers.front.entrypoints=websecure
        - traefik.http.routers.front.tls.certresolver=le    # Optional
TLS via Let's Encrypt
        - traefik.http.services.front.loadbalancer.server.port=80
```

Erläuterung:

- **Image:** Verwenden Sie hier die zuvor nach GHCR gepushten Images (`ghcr.io/...`). Achten Sie darauf, dass `<NAMESPACE>` Ihrem GitHub-Namen entspricht.
- **Backend-Env:** Die Umgebungsvariablen im Backend-Service werden so gesetzt, dass die Anwendung zur Laufzeit die DB erreichen kann. In diesem Beispiel werden Host, Port, Name, User, Passwort der DB gesetzt. (Die Werte entsprechen denen im DB-Service – `DB_HOST=db` verweist im internen Netzwerk auf den Container-Namen des DB-Service.)
- **Netzwerke:** Das Backend ist im internen Netz (für DB-Zugriff) **und** im `traefik-public` Netz, damit Traefik es ansprechen kann. Die Frontend-App (z.B. ein nginx, der eine Web-UI ausliefert) braucht nur im `traefik-public` Netz zu sein, da sie keine direkte DB-Verbindung hat.
- **depends_on:** Stellt sicher, dass der DB-Service gestartet wird, bevor das Backend hochfährt (damit bei Start die DB erreichbar ist).
- **Traefik-Labels:** Diese definieren Routing-Regeln: z.B. wird hier angenommen, dass das Backend über Host `api.meine-domain.de` erreichbar sein soll, und das Frontend über `app.meine-domain.de`. Passen Sie diese Domain-Namen an Ihre tatsächlichen Domains an. Beide Router hören auf den EntryPoint `websecure` (443) mit TLS. Das Label `loadbalancer.server.port` muss auf den **internen Port** des Services zeigen, auf dem die Anwendung läuft (z.B. 8000 beim Backend in unserem Dockerfile, 80 für das nginx-Frontend). Traefik leitet dann an diesen Port im Container weiter.
- **TLS-Zertifikate (Optional):** Die Labels mit `tls.certresolver=le` setzen voraus, dass in Traefik ein Zertifikatslöser "le" (LetsEncrypt) konfiguriert ist. Dies bedingt zusätzliche Traefik-Konfiguration (z.B. ACME Email), was hier nur angedeutet ist. Für den Anfang können Sie die TLS-Option weglassen oder Traefik auf Port 80 (HTTP) testen. In Produktion sollten Sie TLS einrichten.

9. **Deployment des Stacks auf dem Server und Überprüfung**
   Beschreibung: Das vorbereitete `docker-compose.yml` (Stack-Definition) wird nun auf dem Swarm-Manager deployt. Danach werden alle Services als Docker-Swarm-Dienste gestartet. Im Anschluss ist zu prüfen, ob alle Container laufen und die Dienste miteinander kommunizieren (laut Datenfluss im Architekturdiagramm). Außerdem sollte Traefik die Routen korrekt registriert haben. Eventuell müssen Feineinstellungen vorgenommen werden (z.B. Firewall, Domain DNS, etc.) bevor alles produktiv erreichbar ist.

Ziel: Den kompletten Stack auf dem Server zum Laufen bringen – alle Services sollten im Status "running" sein, und ein Testaufruf der Anwendung (Frontend/UI oder API) über Traefik bestätigt die erfolgreiche Kommunikation.
Verantwortlich: DevOps
Durchführung: Kopieren Sie die fertige `docker-compose.yml` auf den Server (oder holen Sie das Git-Repository auf den Server) und deployen Sie den Stack mit einem eindeutigen Namen (z.B. `unityplatform`):

```
docker stack deploy -c docker-compose.yml --with-registry-auth
unityplatform
```

Das `--with-registry-auth` Flag stellt sicher, dass die im Manager hinterlegten Registry-Credentials an die Dienste weitergereicht werden, damit private GHCR-Images gezogen werden können. Prüfen Sie nach dem Deploy den Status aller Services:

```
docker stack services unityplatform    # Zeigt Service-Auflistung mit
Replicas und Status
docker service logs -f unityplatform_backend   # (Beispiel) Logs des
Backend-Service folgen
docker service ps unityplatform_backend        # Zeigt die Task/
Container des Backend-Services
```

Warten Sie einen Moment nach Deployment und stellen Sie sicher, dass für jeden Service die gewünschte Anzahl an Instanzen läuft (bei uns je 1) und keine Crash-Loop oder Fehler auftreten. Kontrollieren Sie `docker service logs` für Fehlermeldungen (z.B. DB-Verbindungsfehler im Backend). Wenn alle Services grün sind, testen Sie die Anwendung von außen: Rufen Sie die konfigurierten URLs in einem Browser oder via `curl` auf (z.B. `https://app.meine-domain.de` für das Frontend, oder einen API-Endpunkt über `https://api.meine-domain.de`). Traefik sollte die Anfragen an die entsprechenden Services weiterleiten. Bei erfolgreicher Antwort ist das Deployment gelungen.

Sollten Probleme auftreten (z.B. ein Service startet nicht), prüfen Sie die Logs und passen Sie ggf. die Konfiguration an. Typische Probleme könnten falsche Umgebungsvariablen, fehlende Netzwerk-Zuordnung oder falsche Image-Namen sein. Nach Korrektur können Sie den Stack mit dem geänderten Compose-File erneut deployen (der Befehl aktualisiert geänderte Services). Damit ist ein **funktionierender Produktionsstart** des "unity-ai-platform-clean" Projekts erreicht: Alle Services laufen im Docker Swarm, sind über Traefik erreichbar und kommunizieren gemäß der vorgesehenen Architektur miteinander.

---

[1] Run Docker Engine in swarm mode | Docker Docs
https://docs.docker.com/engine/swarm/swarm-mode/

[2] [3] [4] Working with the Container registry - GitHub Docs
https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry

[5] Putting labels on services in Docker swarm doesn't work - Traefik v2 - Traefik Labs Community Forum
https://community.traefik.io/t/putting-labels-on-services-in-docker-swarm-doesnt-work/2264