Projektdokumentation M169/M347

07A. Docker-Compose Projekt

Vullnet Ajro, Caspar Schärer

Inhaltsverzeichnis

1. Projektbeschrieb 3
   1.1. Ausgangslage 3
   1.2. Zielsetzung 3
   1.3. Vorgehen 3
   1.4. Technische Umsetzung (Kurzüberblick) 4
2. Arbeitsplanung 5
   2.1. Rollenverteilung und Arbeitsmodus 5
   2.2. Architektur und Containeraufteilung 5
   2.3. Netzwerke und Erreichbarkeit 6
   2.4. Secrets Handling 6
   2.5. Ressourcenbeschränkungen 6
3. Arbeitsjournal 7
4. Testkonzept 8
   4.1. Getestete Dienste 8
   4.2. Randbedingungen (Testumfeld) 8
   4.3. Testmittel und Methode 9
   4.4. Testfälle 10
      TC-01 Nextcloud: Datei erstellen und Persistenz prüfen 10
      TC-02 CVS (Gitea): Commit pushen und Persistenz prüfen 11
      TC-03 MediaWiki: Seite erstellen und Persistenz prüfen 12
      TC-04 Security Check – Datenbank Isolation 13
5. Testprotokoll 14
      Detaillierte Testergebnisse: 14
6. Sicherheitskonzept 15
   6.1. Schutzziele und Risikoanalyse 15
   6.2. Spezifische Gefährdungen und Massnahmen 16
   6.3. Umsetzung Secrets Handling 16

1. Projektbeschrieb

1.1. Ausgangslage

Im Modul M169 haben wir gelernt, Applikationsstacks für Microservices mit Docker Compose zu konfigurieren und als Infrastructure as Code (IaC) bereitzustellen. In diesem Projekt setzen wir dies für ein fiktives Informatik-KMU um.

1.2. Zielsetzung

Ziel des Projekts ist die Bereitstellung einer funktionsfähigen, dokumentierten und versionierten Container-Infrastruktur, die folgende Dienste umfasst:

- **MediaWiki:** Eine firmeninterne Wissensplattform.
- **Nextcloud:** Eine Open-Source-Plattform für Filesharing und Collaboration.
- **Gitea:** Ein firmeninternes CVS (Concurrent Versions System) zur Quellcodeverwaltung.
- **Portainer:** Eine Instanz zur grafischen Überwachung und Verwaltung der Container.

Zusätzlich müssen folgende Anforderungen erfüllt sein:

- **Persistenz:** Dauerhafte Speicherung aller relevanten Daten (Datenbanken, Repositories, Konfigurationen) mittels Docker Volumes.
- **Versionierung:** Speicherung der IaC-Konfigurationsdateien in einem privaten GitHub-Repository.

1.3. Vorgehen

Das Projekt orientiert sich am empfohlenen Vorgehen nach IPERKA;

(Informieren, Planen, Entscheiden, Realisieren, Kontrollieren, Auswerten)

Die Infrastruktur wurde schrittweise aufgebaut.

Nach jedem abgeschlossenen Block wurden die Änderungen mit aussagekräftigen Commit Nachrichten ins GitHub Repository gepusht, um den Arbeitsfortschritt nachvollziehbar zu dokumentieren.

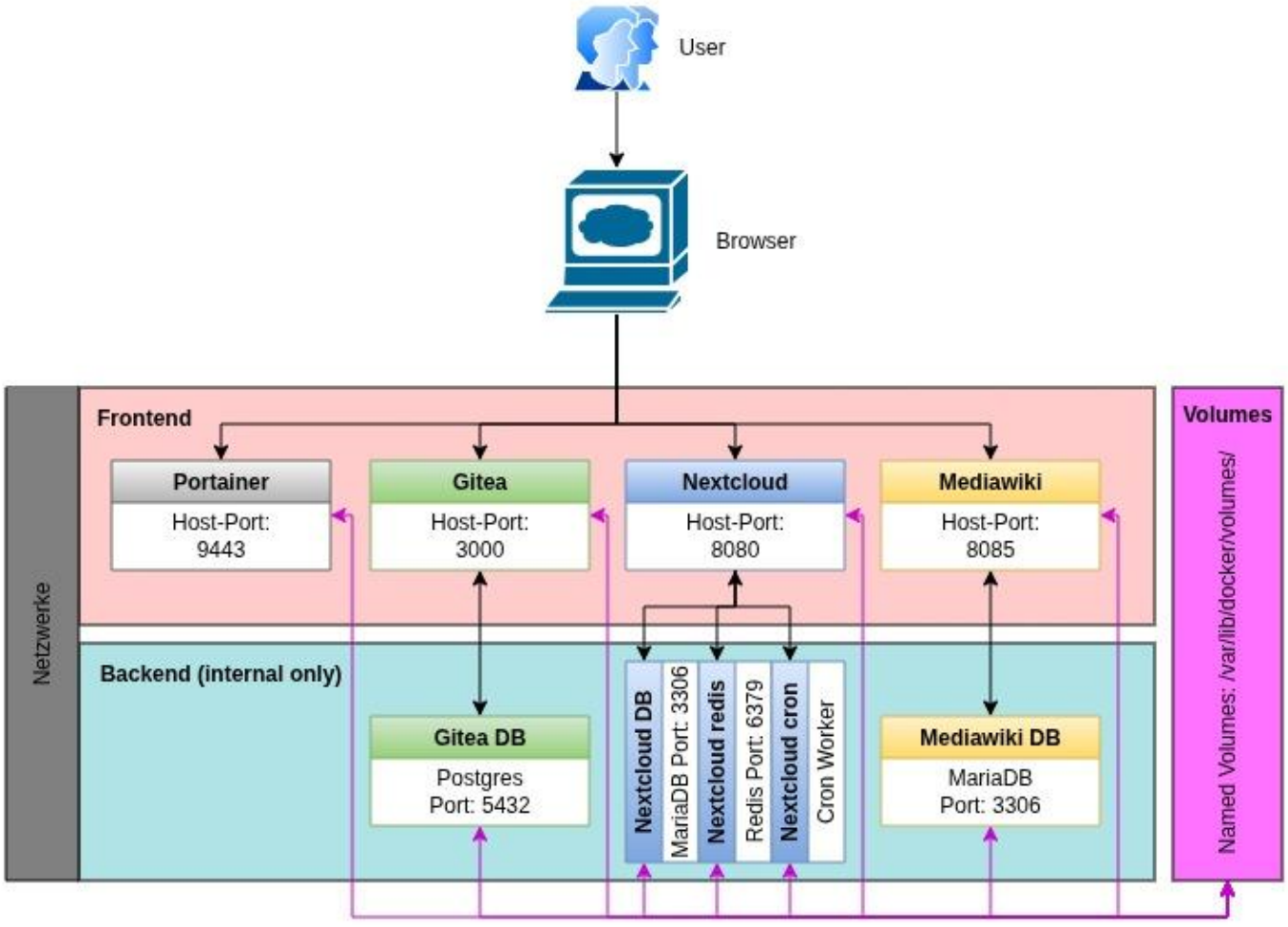
1.4. Technische Umsetzung (Kurzüberblick)

Die Infrastruktur wird über Docker Compose aufgebaut und besteht aus mehreren Containern, die in zwei Netzwerke aufgeteilt sind:

- **frontend Netzwerk:** Dienste, die von Benutzer:innen im Browser direkt erreicht werden (Nextcloud, MediaWiki, Gitea, Portainer)
- **backend Netzwerk (internal):** Dienste, die nur intern erreichbar sind
  →Datenbanken (Postgres, MariaDB)

  →Nextcloud (Redis und -Cronworker)

Zur Persistenz werden Docker Named Volumes eingesetzt. Dadurch bleiben Daten und Konfigurationen auch nach Container Neustarts erhalten. Ressourcenbeschränkungen (CPU, RAM und Swap) wurden gesetzt, um den Betrieb kontrolliert und nachvollziehbar zu gestalten.



Systemübersicht der verwendeten Container, Ports und Netzwerke

2. Arbeitsplanung

2.1. Rollenverteilung und Arbeitsmodus

Um die Stärken im Team optimal zu nutzen, wurden die Hauptverantwortlichkeiten wie folgt aufgeteilt:

- Caspar Schärer: technisch führend (Docker Compose, Services, Persistenz, Ressourcenlimits, Troubleshooting)
- Vullnet Ajro: dokumentationsführend (Dokumentationsstruktur, Arbeitsplanung, Arbeitsjournal, Systemübersicht, Reviews)
- Arbeitsmodus: Peer-Programming und gegenseitige Reviews. Technische Änderungen wurden gemeinsam besprochen, umgesetzt primär durch Caspar, dokumentiert und gegengeprüft primär durch Vullnet.

2.2. Architektur und Containeraufteilung

Die Infrastruktur wurde als modularer Docker Compose Stack realisiert. Es gibt keine geteilten Ressourcen; jeder Dienst verfügt über seine eigenen, dedizierten Hilfscontainer.

Der Application-Stack setzt sich aus folgenden Komponenten zusammen:

- **Applikationen:**
    - **Nextcloud** (Filesharing)
    - **MediaWiki** (Wissen)
    - **Gitea** (CVS). *Entscheid:* Wir haben uns bewusst für **Gitea** (statt GitLab oder Gogs) entschieden, da es wesentlich ressourcenschonender arbeitet und auf der verfügbaren Hardware deutlich performanter läuft als schwergewichtige Alternativen.
- **Datenbanken:** Jede Applikation nutzt eine eigene Datenbank-Instanz:
    - **MariaDB:** Ein separater Container für Nextcloud und ein eigener Container für MediaWiki.
    - **PostgreSQL:** Ein dedizierter Container für Gitea.
- **Hilfsdienste:** Redis (Caching) und Cron (Hintergrundjobs) sind als spezifische Container exklusiv der Nextcloud zugeordnet.

2.3. Netzwerke und Erreichbarkeit

Um die Angriffsfläche zu minimieren, wurden zwei isolierte Docker-Netzwerke konfiguriert:

1. **frontend:** In diesem Netzwerk befinden sich die Web-Dienste. Hier sind die Ports auf den Host gemappt, damit Benutzer darauf zugreifen können:
    - Nextcloud: Port 8080
    - Gitea: Port 3000
    - MediaWiki: Port 8085
    - Portainer: Port 9000 (bzw. 9443)
2. **backend:** Hier kommunizieren die Applikationen mit ihren Datenbanken und Hilfsdiensten (Redis). Dieses Netzwerk hat keine Verbindung nach aussen (nur Service-Port mapping), was den direkten Zugriff auf die Datenbanken verhindert.

2.4. Secrets Handling

Sensible Daten (wie Datenbank-Passwörter) werden nicht im Klartext in der docker-compose.yml gespeichert.

- **Lokale Speicherung:** Passwörter liegen in einer lokalen Datei namens .env.
- **Git-Sicherheit:** Die Datei .env ist im .gitignore eingetragen, damit keine echten Secrets im Repository landen.
- **Reproduzierbarkeit:** Eine Datei compose/.env.example (mit Platzhaltern) wird versioniert, um anderen Entwicklern die Struktur aufzuzeigen.

2.5. Ressourcenbeschränkungen

Um die Stabilität des Host-Systems zu gewährleisten, wurden für alle Container Limits definiert. Dabei wurde folgende Priorisierung vorgenommen:

- **Arbeitsspeicher (RAM) und Swap (Priorität 1):** Die Begrenzung des Speichers ist für die Stabilität am wichtigsten. Jeder Container hat ein festes RAM-Limit. Zusätzlich wurde der Swap (memswap_limit) strikt begrenzt. Dies verhindert, dass ein einzelner Container bei einem Speicherleck (Memory Leak) den gesamten Server durch massives Auslagern (Swapping) lahmlegt oder zum Absturz bringt.
- **Prozessor / CPU (Priorität 2):** Ergänzend wurde die CPU-Nutzung beschränkt. Dies hat zweite Priorität, da eine hohe CPU-Last das System zwar verlangsamen, aber selten zum Absturz bringen kann. Die Limits sorgen hier primär für eine faire Verteilung der Rechenleistung zwischen den Diensten.

3. Arbeitsjournal

Datum	Zeit (h)	Wer	Tätigkeit	Resultat
09.01.26	2	Alle	Initialisierung Projekt & Git Repository	Repository erstellt, .gitignore gesetzt
14.01.26	6	Caspar	Setup Gerüst, Portainer, Gitea, Nextcloud, Wiki	Alle Services laufen inkl. Datenbanken & Persistenz
14.01.26	2	Caspar	Implementierung Ressourcenlimits (CPU/RAM)	Limits in docker-compose.yml definiert
15.01.26	1	Caspar	Cleanup & Dokumentation (README)	Code bereinigt, erste Anleitung erstellt
20.01.26	1	Caspar	Security: Swap-Limits (memswap) ergänzt	memswap_limit für alle Container gesetzt
22.01.26	3	Vullnet	Dokumentationsaufbau, Inhaltsverzeichnis	Struktur der Doku steht
23.01.26	4	Alle	Finalisierung Doku, Testprotokolle, Review	Projekt abgeschlossen & abgabebereit

4. Testkonzept

Ziel des Testkonzepts ist der Nachweis, dass die drei geforderten Hauptdienste funktionsfähig und im Browser erreichbar sind. Zusätzlich wird verifiziert, dass erstellte Inhalte nach einem Neustart der Container weiterhin vorhanden sind (Persistenz).

4.1. Getestete Dienste

- **Nextcloud:** Filesharing und Collaboration Plattform.
- **Gitea (CVS):** Firmeninterne Quellcode-Verwaltung.
- **MediaWiki:** Firmeninterne Wissensplattform.

4.2. Randbedingungen (Testumfeld)

- **System**: Linux-Rechner (Laptop oder VM).
- **Betriebssystem**: Ubuntu 24.04.3 LTS (Release 24.04, Codename noble).
- **Software**:
    - **Docker Engine:** Docker Version 29.0.2 (Build 8108357).
    - **Docker Compose:** Docker Compose Version v2.40.3.
    - **Git:** Git Version 2.43.0.

- **Benutzerrechte:** Ausführung als normaler Benutzer mit Docker-Rechten (Mitglied der Gruppe docker), kein permanenter Root-Login erforderlich.

- **Projektbasis**: Privates GitHub-Repository, Projektordner m169/Abschlussprojekt/compose.

- **Konfiguration**:
    - compose/.env.example ist versioniert und dient als Vorlage.
    - compose/.env wird lokal erstellt (cp .env.example .env) und enthält Passwörter/Secrets.
    - compose/.env ist im .gitignore und wird nicht ins Repository gepusht.

- **Startbefehle:**

```
docker compose pull
docker compose up -d
```

- **Netzwerk:** Zugriff lokal über Browser auf localhost, kein Reverse Proxy im Einsatz.
- **Erreichbarkeit (Ports):**
    - **Nextcloud**: http://localhost:8080
    - **Gitea**: http://localhost:3000
    - **MediaWiki**: http://localhost:8085

- **Persistenz:** Speicherung erfolgt über Docker Named Volumes (Applikationsdaten und Datenbanken). Persistenz Tests erfolgen durch Neustart relevanter Container und anschliessende Verifizierung im Webinterface.
- **Erreichbarkeit (Ports):**
    - Nextcloud: http://localhost:8080
    - Gitea: http://localhost:3000
    - MediaWiki: http://localhost:8085

4.3. Testmittel und Methode

Testmittel:

- Webbrowser
- Terminal
- Docker Compose CLI

Methode:

- **Funktionstests:** Überprüfung der Grundfunktionen im Web-UI (Blackbox-Test).
- **Persistenz Tests:** Erstellen von Daten, Neustart der Container (docker compose restart) und anschliessende Verifizierung der Daten.

4.4. Testfälle

TC-01 Nextcloud: Datei erstellen und Persistenz prüfen

Ziel: Nachweis, dass Nextcloud läuft und Benutzerdaten nach einem Container-Neustart erhalten bleiben.

Voraussetzungen:

- Nextcloud Stack läuft (nextcloud, nextcloud-db, nextcloud-redis, nextcloud-cron)
- Admin-Login existiert

Schritte:

1. Nextcloud öffnen: http://localhost:8080
2. Als Admin anmelden.
3. In Dateien eine neue Textdatei erstellen: persistenz-test-nextcloud.txt
4. Datei öffnen, Text einfügen: “Test am <Datum> um <Uhrzeit>”, speichern.
5. Container neu starten:

```
docker compose restart nextcloud nextcloud-db nextcloud-redis nextcloud-cron
```

6. Browser neu laden (Reload).
7. Prüfen, ob die Datei noch da ist und der Inhalt unverändert ist.

Erwartetes Resultat:

- Datei existiert nach Neustart weiterhin.
- Inhalt ist unverändert.

TC-02 CVS (Gitea): Commit pushen und Persistenz prüfen

Ziel: Nachweis, dass Gitea mit Datenbank funktioniert und Repositories persistent sind.

Voraussetzungen:

- gitea und gitea-db laufen
- Ein Benutzerkonto existiert

Schritte:

1. Gitea öffnen: http://localhost:3000
2. Anmelden.
3. Repo erstellen: firma-demo (falls noch nicht vorhanden).
4. Lokal Testrepo erstellen:

```
mkdir -p ~/tmp/gitea-test
cd ~/tmp/gitea-test
git init
git branch -M main
echo "Hallo Gitea" > README.md
git add README.md
git commit -m "test: first commit"
```

5. Remote setzen und pushen (USERNAME anpassen):

```
git remote add origin http://localhost:3000/<USERNAME>/firma-demo.git
git push -u origin main
```

6. In Gitea Web UI prüfen, ob Commit sichtbar ist.
7. Container neu starten:

```
docker compose restart gitea gitea-db
```

8. Web UI neu laden und prüfen, ob Repo und Commit weiterhin sichtbar sind.

Erwartetes Resultat:

- Commit ist im Web UI sichtbar.
- Nach Neustart sind Repo und Commit weiterhin vorhanden.

TC-03 MediaWiki: Seite erstellen und Persistenz prüfen

Ziel: Nachweis, dass MediaWiki funktioniert und Seiten nach Neustart erhalten bleiben.

Voraussetzungen:

- mediawiki und mediawiki-db laufen
- MediaWiki ist fertig eingerichtet (LocalSettings.php ist im Container vorhanden)

Schritte:

1. MediaWiki öffnen: http://localhost:8085
2. Anmelden.
3. Neue Seite erstellen, z.B. “ProjektTestseite”:
    - Inhalt: “Testseite erstellt am <Datum> um <Uhrzeit>”
    - Speichern.
4. Seite aufrufen und prüfen, ob der Inhalt sichtbar ist.
5. Container neu starten:

```
docker compose restart mediawiki mediawiki-db
```

6. MediaWiki neu laden und die Seite “ProjektTestseite” erneut aufrufen.
7. Prüfen, ob die Seite und der Inhalt weiterhin vorhanden sind.

Erwartetes Resultat:

- Seite existiert nach Neustart weiterhin.
- Inhalt ist unverändert.
- MediaWiki ist weiterhin erreichbar.

TC-04 Security Check – Datenbank Isolation

Ziel: Verifikation, dass Datenbank-Container von aussen nicht direkt erreichbar sind.

Voraussetzungen:

- Alle Container laufen.

Schritte:

1. Versuch, die Datenbank direkt anzusprechen (z.B. via telnet localhost 3306 oder 5432).Anmelden.
2. Prüfung der Ausgabe auf Port-Mappings.

```
docker ps
```

Erwartetes Resultat:

- Verbindung wird abgelehnt; keine Ports der Datenbanken sind auf den Host gemappt.

5. Testprotokoll

ID	Testfall	Erwartetes Resultat	Status
TC-01	Nextcloud Persistenz Datei hochladen & Container neu starten	Datei ist nach dem Neustart unverändert vorhanden.	PASS
TC-02	Gitea Repository Repo erstellen, Commit pushen & DB neu starten	Repository und Commit sind nach Neustart vorhanden.	PASS
TC-03	MediaWiki Inhalt Neue Seite erstellen & Wiki neu starten	Die Seite ist nach dem Neustart abrufbar.	PASS
TC-04	Security Check Ping auf Datenbank von extern	Datenbank-Container sind von aussen nicht erreichbar.	PASS

Detaillierte Testergebnisse:

- **Zu TC-01 (Nextcloud):** Die Datei persistenz-test-nextcloud.txt wurde erfolgreich erstellt. Nach dem Neustart des gesamten Nextcloud-Stacks war die Datei inklusive Inhalt (“Test am...”) weiterhin verfügbar.
- **Zu TC-02 (Gitea):** Das Repository firma-demo wurde erstellt. Ein lokaler Commit konnte erfolgreich gepusht werden. Nach dem Neustart der Container gitea und gitea-db war der Commit im Web-Interface unverändert sichtbar.
- **Zu TC-03 (MediaWiki):** Die Seite „ProjektTestseite“ wurde angelegt. Nach dem Neustart der MariaDB und des MediaWiki-Containers war die Seite weiterhin mit dem korrekten Inhalt abrufbar.
- **Zu TC-04 (Security):** Ein direkter Zugriff auf die Datenbank-Ports war nicht möglich, da diese Ports im Docker-Netzwerk backend isoliert sind und nicht an den Host weitergeleitet werden.

6. Sicherheitskonzept

6.1. Schutzziele und Risikoanalyse

In Anlehnung an den BSI-Baustein **SYS.1.6 (Containerisierung)** verfolgen wir für unser KMU-Projekt drei zentrale Schutzziele:

1. **Vertraulichkeit:** Schutz sensibler Firmendaten in Nextcloud und Gitea vor unbefugtem Zugriff durch strikte Isolation der Netzwerke.
2. **Integrität:** Sicherstellung, dass Quellcode (Gitea) und Wiki-Einträge nicht unautorisiert verändert werden können.
3. **Verfügbarkeit:** Schutz vor Systemausfällen durch Ressourcenlimits, um Denial-of-Service (DoS) durch einzelne Container zu verhindern.

Gefährdung	Auswirkung	Eintrittswahrscheinlichkeit	Risiko
Container Breakout	Existenzbedrohend	Gering	**Mittel**
Unbefugter Datenbank-Zugriff	Hoch	Mittel	**Hoch**
Kompromittierung Portainer	Existenzbedrohend	Mittel	**Hoch**
Ressourcen-Erschöpfung	Mittel	Mittel	**Mittel**

6.2. Spezifische Gefährdungen und Massnahmen

Wir haben folgende Gefahren identifiziert und durch technische Massnahmen minimiert:

- **Gefahr: Container-Breakout** (Ausbruch auf das Host-System)
    - *Massnahme:* Wir führen keine Container im privileged-Modus aus. Wo immer möglich, nutzen wir Images, die nicht als root-User laufen.
- **Gefahr: Unautorisierter Zugriff auf Konfigurationen**
    - *Massnahme:* Konfigurationsdateien und Docker-Volumes sind über Linux-Dateiberechtigungen auf dem Host-System geschützt.
- **Gefahr: Kompromittierung via Portainer (Docker Socket)**
    - *Massnahme:* Portainer hat Zugriff auf den Docker-Socket (/var/run/docker.sock), was faktisch Root-Rechten entspricht. Daher darf Portainer **nur** in einem abgesicherten Netzwerk (z.B. VPN oder localhost) erreichbar sein und muss durch ein **starkes Admin-Passwort** (mind. 20 Zeichen) geschützt werden.
- **Gefahr: Schwachstellen in der Virtualisierung**
    - *Massnahme:* Regelmässige Updates der Docker-Engine und ausschliessliche Nutzung von offiziellen, signierten Images (Verified Publisher).

6.3. Umsetzung Secrets Handling

Um Passwörter in Docker-Compose-Files auf GitHub zu vermeiden (ein kritisches Sicherheitsrisiko), haben wir folgende Architektur implementiert:

1. **Vermeidung von Klartext:** Es stehen keine Passwörter direkt in der docker-compose.yml.
2. **Externalisierung:** Nutzung von Umgebungsvariablen über eine .env-Datei.
3. **Git-Hygiene:** Die Datei .env wurde via .gitignore vom Repository ausgeschlossen. Eine Datei compose/.env.example wurde ohne reale Passwörter hochgeladen, um die Struktur für Dritte (z.B. Dozenten) ersichtlich zu machen.