Duale Hochschule Baden-Württemberg Mannheim

Projektberichtarbeit

Drei-Schicht Anwendung

Studiengang Informatik

Studienrichtung Angewandte Informatik

Verfasser(in):	Moritz Werr, Phil Richter, Max Stege
Matrikelnummer:	5401527
Matrikelnummer:	4164342
Matrikelnummer:	7285772
Kurs:	TINF22AI2
Dozent:	Jochen Kluger
Bearbeitungszeitraum:	01.10.2024 – 20.12.2024

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Titel "*Drei-Schicht Anwendung*" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Quelltextverzeichnis

1 Einleitung (Moritz Werr)

Im Rahmen dieses Projektes wurde eine Anwendung in drei Schichten aufgeteilt mithilfe von Docker und Docker-Compose. Diese wurde im Rahmen eines vorherigen Semesters implementiert als eine monolithische Serveranwendung, obwohl diese jedoch eigentlich aus mehreren Komponenten besteht. Generell besteht diese aus drei Teilen: dem Webserver und der Programmierlogik für die Anwendung, einem Datenbankserver für die persistente Speicherung der Daten und einer TLS-Verschlüsselung für die Sicherheit der Datenübertragung. Vorher liefen diese alle auf einem Server, es war eine monolithische Architektur. Nun wird diese in ihren einzelnen Funktionalitäten aufgespalten, hin zu einer Mikroservicearchitektur. Jede einzelne Funktionalität wird in einen eigenen Softwarecontainer übertragen und laufen isoliert auf der gleichen Maschine, indem sie sich den Betriebsystemkernel teilen [5, Vgl. S. 152444]. Dieser Prozess nennt sich Containerisierung.

Durch die Umwandlung in eine Mikroservicearchitektur wird ein hoher Grad an Flexibilität gewonnen. Die einzelnen Komponenten arbeiten unabhängig voneinander und ergeben gemeinsam ein funktionierendes System [2, Vgl. S.1f]. Dafür muss die Anwendung jedoch in manchen Fällen umgeschrieben werden. Durch den Umbau zu einer Mikroservicearchitektur, wird oftmals die Komplexität der Anwendung erhöht [4, Vgl. S.10], weswegen noch weitere Werkzeuge zur Verwaltung der Anwendung benötigt werden. Hilfswerkzeuge zur Verwaltung und Konfiguration der Anwendung werden benötigt.

Weiterhin sind nicht nur die einzelnen Komponenten austauschbar, sondern können nun auch weitere Funktionalitäten durch die Containerisierung erreicht werden. Es können sehr einfach Healthchecks implementiert werden, die überprüfen, ob einzelne Komponenten noch richtig funktionieren [3]. Wenn dies nicht mehr der Fall ist, können die einzelnen Komponenten der drei Schichten neugestartet werden [3]. Dafür muss definiert werden wie diese miteinander zusammenarbeiten. Wenn nun ein einzelner Container Probleme hat, wird dies erkannt und er kann automatisch neugestartet werden. Weiterhin sorgt dies auch dafür, dass Container in der richtigen Reihenfolge gestartet werden [3].

Im Rahmen dieses Projektes wurden Docker und Docker-Compose verwendet aufgrund ihrer guten Dokumentation und großen Anzahl an Features. Kernbestandteil war Docker zur Containerisierung und Docker-Compose zur Verwaltung der Container. Es musste ein neues Dockerfile für den Webserver erstellt werden, zwei vorhandene Containerimages wurden für die TLS-Verschlüsselung (traefik) bzw. die Datenspeicherung (mariadb) verwendet und ein Initialisierungsskript für die Datenbank wurde erstellt.

2 Docker als Softwarecontainer (Moritz Werr)

Für die Containerisierung wurde Docker verwendet. Softwarecontainer bieten viele Vorteile gegenüber traditioneller Virtualisierung. Traditionelle Virtualisierung wird verwendet um Ressourcen effizienter zu nutzen. Dabei werden viele schwache Maschinen in einer großen Maschine konsolidiert [5, Vgl. S.152443]. Diese teilen sich alle die gleichen Hardwareressourcen, welche über den Hypervisor auf die einzelnen virtuellen Maschinen verteilt wird. Allerdings teilen sich virtuelle Maschinen nicht ihre Softwarekomponenten. Dadurch wird oftmals der gleiche Programmcode öfters im Arbeitsspeicher vorkommen. Softwarecontainer funktionieren in dieser Hinsicht anders. Sie teilen sich normalerweise den Kernel mit dem Containerhost [5, Vgl. S.152444]. Diese sind dadurch um einiges leichtgewichtiger und schneller als traditionelle Virtualisierung [1, Vgl. S.137]. Aufgrund dieser Leichtgewichtigkeit und Geschwindigkeit lassen sich Docker-Anwendungen mit Orchestrierungswerkzeugen wie Kubernetes auch sehr leicht horizontal und vertikal skalieren [5, Vgl. S.152458].

Weiterhin sind Anwendungen innerhalb von Docker-Containern auch sehr portabel, da ein Container nicht vom Hostsystem abhängig ist, sondern von dessen Docker-Image. Alle Abhängigkeiten (Dependencies) sind in einem einzelnen Image gebündelt [1, Vgl. S.137]. Dadurch sind containerisierte Anwendungen hoch portabel. Sie können auf jeder Maschine laufen, die eine kompatible Docker-Engine hat. Nicht nur läuft die Anwendung auf sehr vielen Maschinen, sondern es läuft auch überall die gleiche Version. Das Image sorgt für einen hohen Grad an Konsistenz und Wiederholbarkeit.

Außerhalb der hohen Effizienz und Portabilität von Softwarecontainern, bieten diese auch noch einen gewissen Grad an Sicherheit. Obwohl virtuelle Maschinen einen höheren Grad an Sicherheit bieten durch getrennte Kernel [5, Vgl. S.152455], bieten Container einen guten Schutz durch Prozessisolierung, Dateisystemisolierung, Ressourcenlimitierung und Netzwerkisolation [5, Vgl. S.152444ff]. Dafür werden Funktionalitäten des Linux Kernels verwendet. Primär wären dies CGroups, Namespaces und MAC-Systemen (Mandatory Access Control) wie SELinux oder Apparmor. Cgroup sorgen dabei für die Resourcenlimitierung, Namespaces für die Prozess- und Netzwerkisolation und Apparmor für die Dateisystemisolierung.

Durch die Nutzung von Docker ist die Anwendung sehr performant, portabel und auch noch gut abgesichert.

3 TLS-Proxy (Moritz Werr)

Im der Docker-Compose-Datei `docker-compose.yml` definiert folgender Abschnitt den Traefik-Container:

```
1
2  services:
3    traefik:
4      image: traefik:v2.10
5      command:
6        - "--api.insecure=true"
7        - "--providers.docker=true"
8        - "--entrypoints.web.address=:80"
9        - "--entrypoints.websecure.address=:443"
10       - "--providers.file.filename=/dynamic/traefik_dynamic.
           yml"
11     ports:
12       - "80:80"
13       - "443:443"
14       - "8080:8080"
15     volumes:
16       - "/var/run/docker.sock:/var/run/docker.sock"
17     depends_on:
18       db:
19         condition: service_healthy
20         restart: true
21       webserver:
22         condition: service_healthy
23         restart: true
```

Quelltext 3.1: `docker-compose.yml`

Traefik übernimmt hier die TLS-Verschlüsselung für die Webanwendung. Dafür gibt es bereits ein existierendes Containerimage, weswegen dieses nur noch konfiguriert werden muss. Es wird ebenfalls in der gleichen Docker-Compose-Datei definiert.

Zuerst wird das Image ausgewählt für den Service mit dem Namen `traefik` genannt. Danach werden noch weitere Kommandos erwähnt, damit Traefik richtig konfiguriert ist . Primär muss TLS aktiviert werden und die entsprechenden HTTP- und HTTPS-Ports freigegeben werden.

In unserem Fall wurde das Standardzertifikat von Traefik verwendet, weil selbstsignierte Zertifikate hinterlegen durchaus schwierig ist. Bei uns handelt es um einen Prototypen und nicht um ein produktionsreifes Endergebnis.

Als nächstes werden Port 80 und 443 für die Kommunikation freigegeben. Weiterhin wird der Port 8080 für das Managen des Containers freigegeben. Der normale Webservercontainer ist nicht außerhalb des Docker-Netzwerkes erreichbar, weswegen über den Traefik-Container kommuniziert werden muss.

Weiterhin wird der Docker-Socket vom Hostsystem in den Container gereicht, damit der Traefik-Container den Webservercontainer finden kann (Service Discovery).

Als letztes werden die Healthchecks für den Traefik-Container definiert. Da eine TLS-Verschlüsselung nicht ohne Applikationsserver funktioniert, mus dieser funktionieren. Weiterhin kann der Applikationsserver nicht ohne die Datenbank funktionieren. Der Traefik-Container ist nur indirekt von diesem abhängig, trotzdem wird dieser hier erwähnt, um die Abhängigkeit zu verdeutlichen.

4 Bedienung der Anwendung (Moritz Werr)

Bevor die Anwendung gestartet werden kann, muss zuerst das Image des Webservers gebaut werden. Dies wird normalerweise mit dem Befehl `docker build` gemacht. Der folgende Befehl baut das Image, wenn man sich im Wurzelverzeichnis dieses Projektes befindet:

```
docker build . -t webserver
```

bzw.

```
docker compose build
```

Nachdem das Image gebaut wurde, kann die Anwendung gestartet werden. Im Rahmen unseres Projektes haben wir die Emailauthentifizierung größtenteils hartkodiert. Die Datei `.env` beinhaltet das Passwort für den Emailversand und ist der Einfachheit in der ZIP-Datei enthalten, jedoch nicht auf Github. Wenn ein eigener Emailprovider verwendet werden soll, muss die Datei `msmtp.rc` im Projektordner angepasst werden.

Die Anwendung kann durch individuelle `docker run`-Befehle gestartet werden, wird aber wegen der großen Anzahl an Parametern nicht bevorzugt. Stattdessen wird mit Docker-Compose die bereits konfigurierte Anwendung gestartet:

```
docker compose up -d
```

Damit der Befehl funktioniert muss im Wurzelverzeichnis des Projektes der Befehl ausgeführt werden. Sonst muss der Pfad zu Datei explizit angegeben werden:

```
docker compose -f /path/to/project/folder/docker-compose.yml up -d
```

Das Bauen des Images kann hier auch vollführt werden:

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

Damit wird die Anwendung im Hintergrund gestartet und erstellt auch noch automatisch die definierten Docker-Volumes. Weiterhin wird ein eigenes Docker-Netzwerk für die Anwendungscontainern erstellt. Dieses kann über die Hostports 80, 443 und 8080 erreicht werden. Die Anwendung sollte nun über folgenden Link erreichbar sein.

Dort können Sie sich mit den Anmeldeinformationen aus der SQL-Datei anmelden (Standardmäßig: *admin* bzw. *user* mit dem Passwort *Test12345!*).

Anschließend kann die Anwendung mit dem folgenden Befehl wieder beendet werden:

```
docker compose down
```

Bibliography

[1] Felipe Bedinotto Fava et al. "Assessing the Performance of Docker in Docker Containers for Microservice-Based Architectures". In: *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). ISSN: 2377-5750. Mar. 2024, pp. 137–142. DOI: 10.1109/PDP62718.2024.00026. URL: https://ieeexplore.ieee.org/document/10495554 (visited on 12/20/2024).

[2] Martin Fowler and James Lewis. "Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr". In: *Objektspektrum* 1.2015 (2015), pp. 14–20.

[3] *Services top-level elements*. Docker Documentation. 100. URL: https://docs.docker.com/reference/compose-file/services/ (visited on 12/20/2024).

[4] Ruoyu Su and Xiaozhou Li. "Modular Monolith: Is This the Trend in Software Architecture?" In: *2024 IEEE/ACM International Workshop New Trends in Software Architecture (SATrends)*. 2024 IEEE/ACM International Workshop New Trends in Software Architecture (SATrends). Apr. 2024, pp. 10–13. URL: https://ieeexplore.ieee.org/document/10669865 (visited on 12/20/2024).

[5] Junzo Watada et al. "Emerging Trends, Techniques and Open Issues of Containerization: A Review". In: *IEEE Access* 7 (2019). Conference Name: IEEE Access, pp. 152443–152472. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2945930. URL: https://ieeexplore.ieee.org/abstract/document/8861307 (visited on 12/20/2024).