Projektdokumentation Homeserver

Thilo Wendt

12. Januar 2020

Zusammenfassung

Inhaltsverzeichnis

1 Einleitung **4**

2 Motivation **5**

3 Anforderungen **6**

4 Projektdurchführung **8**
4.1 Hardware 8
4.2 Verwendete Basis-Software 8
4.2.1 Betriebssystem 8
4.2.2 Containervirtualisierung mit Docker 9
4.2.3 Dateisystem 10
4.2.4 Automatisiertes Backup 11
4.3 Softwarearchitektur 11
4.3.1 Deployment der Container und der Anwendungen 11
4.3.2 Starten der Docker-Container 13
4.3.3 Interne Kommunikation der Container 13
4.4 Webdesign mit Django 15

Literatur **16**

A Docker-Compose Dateien **17**

1 Einleitung

Träumt nicht jeder an Informationstechnik interessierte Mensch von der allumfassenden sicheren und unbegrenzten Cloud-Lösung für einen intuitiven Workflow über alle Geräte und Plattformen hinweg? Der Autor möchte sich diesen Traum erfüllen und seinen eigenen Webserver betreiben, um dort verschiedene Dienste anbieten zu können. Aber Halt! Ist es nicht viel einfacher, diese Lösung an extern zu vergeben und die Dienste von Apple, Google und wie sie nicht alle heißen zu nutzen? Einfacher in jedem Fall, doch möchte man seine Daten nicht mit einem internationalen Konzern teilen, so führt kein Weg an der selbst gebauten Lösung vorbei. Des Weiteren ist es mit den heute verfÃ¼gbaren Open-Source-Werkzeugen einfach geworden, einen eigenen Webserver zu betreiben und diesen universell zu nutzen. Beispielsweise ist auch die Gestaltung eines Webauftritts mit WordPress einfach realisierbar.

Im Folgenden wird zunächst das Problem näher erläutert und die möglichen bereits vorhandenen Lösungen mit der geplanten selbst betriebenen Lösung verglichen. Darauf aufbauend wird eine grobe Systemarchitektur skizziert, sowie das Projekt in Arbeitspakete unterteilt.

2 Motivation

Spätestens seitdem wir angefangen haben, täglich mehrere Computer verschiedener Größe vom Handy bis zur Full-Size-Workstation zu nutzen, haben wir uns bezüglich des Austausches von Informationen und Dateien einige Probleme eingehandelt: Ohne die Nutzung einer Cloud-Lösung ist es schwierig, den automatisierten Austausch zwischen den Geräten zu realisieren. Das Ergebnis ist oft doppelte und inkonsistente Datenhaltung auf unterschiedlichen Medien (USB-Stick, Handy, Laptop etc.). Die grundlegenden Probleme, zu denen im vorliegenden Projekt Lösungen erarbeitet werden sollen, lassen sich auf die folgenden drei Stichpunkte reduzieren.

1. Konsistenz: Wie kann ein einheitlicher Versionsstand zwischen den Geräten hergestellt werden?
2. Verfügbarkeit: Wie kann jedem Gerät zu jedem Zeitpunkt die vollständige Datenbasis zur Verfügung gestellt werden?
3. Vertraulichkeit: Es ist nicht gewünscht, persönliche Daten mit multinationalen Konzernen zu teilen. Wie kann dieses Ziel erreicht werden?

Die Lösung für Problem 1 und 2 scheint wie bereits angedeutet einfach: Die Nutzung einer Cloud-Lösung als verbindende Instanz zwischen allen Geräten. Hierzu gibt es bereits fertige Lösungen wie z.B. Google Drive, iCloud, Dropbox, OneDrive und viele mehr. All diese Fertiglösungen haben jedoch den Nachteil, dass der gesamte persönliche Datenbestand auf einen externen (amerikanischen) Server ausgelagert wird und die dahinter stehenden Unternehmen damit machen können, was sie möchten. Ist dies für den Nutzer akzeptabel steht der Nutzung einer solchen Cloud-Lösung nichts mehr im Wege. Ist man jedoch daran interessiert, selbst über seine Daten zu verfügen, muss etwas mehr Aufwand getrieben werden.

Auch für Home-Clouds gibt es fertige Lösungen. Zu nennen ist hierbei die Serverhardware von QNAP und Synology, welche eine Plug-and-Play-Lösung darstellen. Gegen die Nutzung einer solchen Plattform sprechen jedoch folgende Gründe:

- Preis-Leistungs-Verhältnis: Ein Synology Network-Attached-Storage-Server (NAS) mit einem Dual-Core Prozessor von Intel und 2GB RAM kostet ca. 450€. Für den selben Preis ist es möglich einen Server aus gebrauchten Teilen mit einem Hexa-Core-Xeon Prozessor und 64GB RAM zusammenzustellen.
- Unflexibel: Die Lösungen von QNAP und Synology sind *ausschließlich* als NAS konzipiert. Ein selbst aufgesetzter Server kann z.B. auch als Plattform für ein GitLab ein WordPress-Blog oder Ähnliches dienen.
- Begrenzte Erweiterbarkeit: Die Baugröße eines typischen Plug-and-Play-NAS erlaubt keine größeren Upgrades bezüglich Prozessor, RAM und Massenspeicher.

Die Motivation, eine eigene Plattform aufzusetzen, ist damit begründet. Im folgenden Kapitel wird kurz die Hard- und Software-Architektur beschrieben.

3 Anforderungen

Die Anforderungen an das System seien im Use-Case-Diagramm aus Abbildung 3.1 aufgeführt.

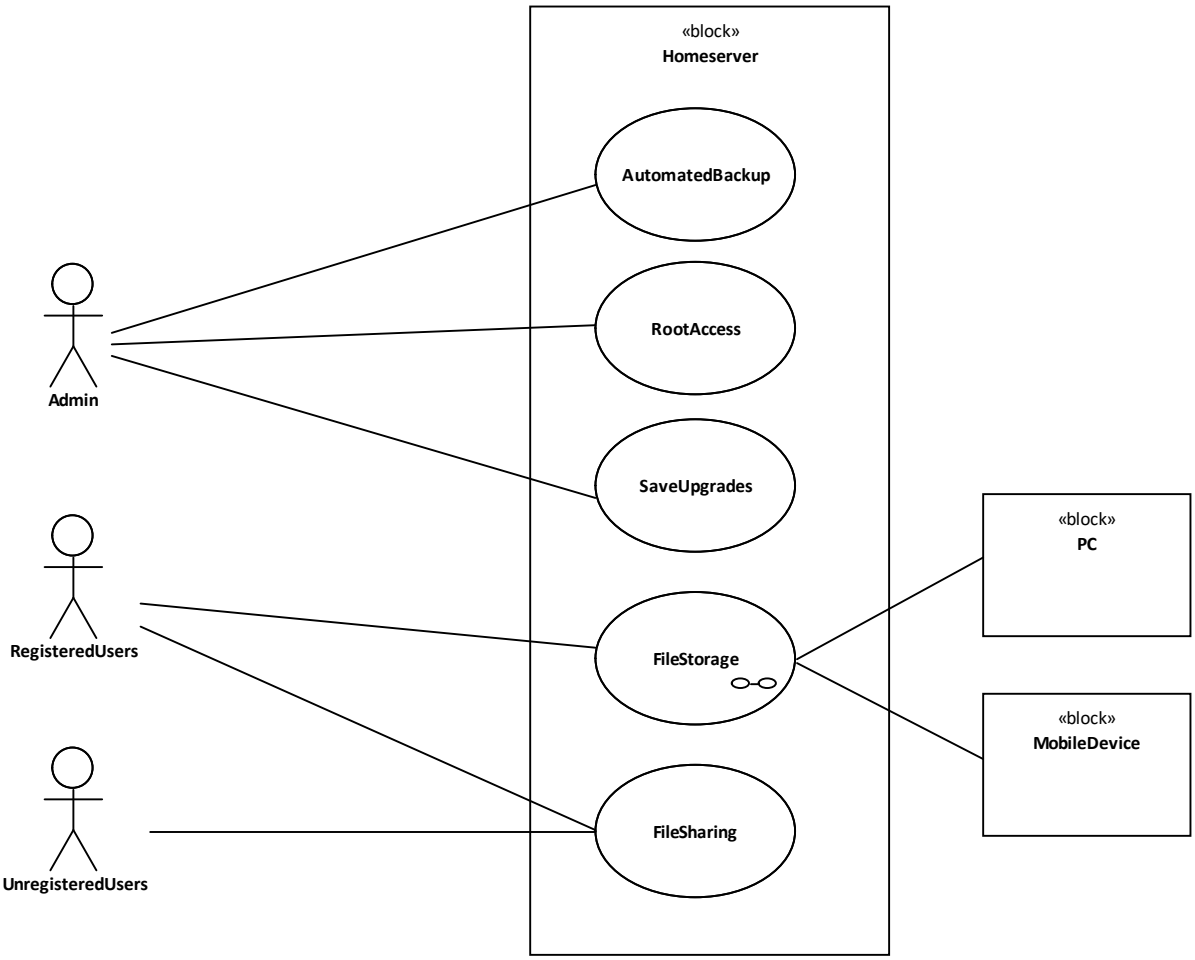

Abbildung 3.1: Use-Case-Diagramm der Anforderungen an das System „Homeserver" gegenüber regestrierten und nicht registrierten Benutzern, sowie dem Administrator und der potentiell einsetzbaren Geräte

Besonders der Use-Case „SaveUpgrade" war bei der Durchführung des Projektes ein besonderer Schwerpunkt. Die Containervirtualisierung mit Docker erfüllt diese Anforderung. Der Anwendungsfall „FileStorage" lässt sich noch weiter verfeinern. Das separate Diagramm ist in Abbildung 3.2 zu finden.

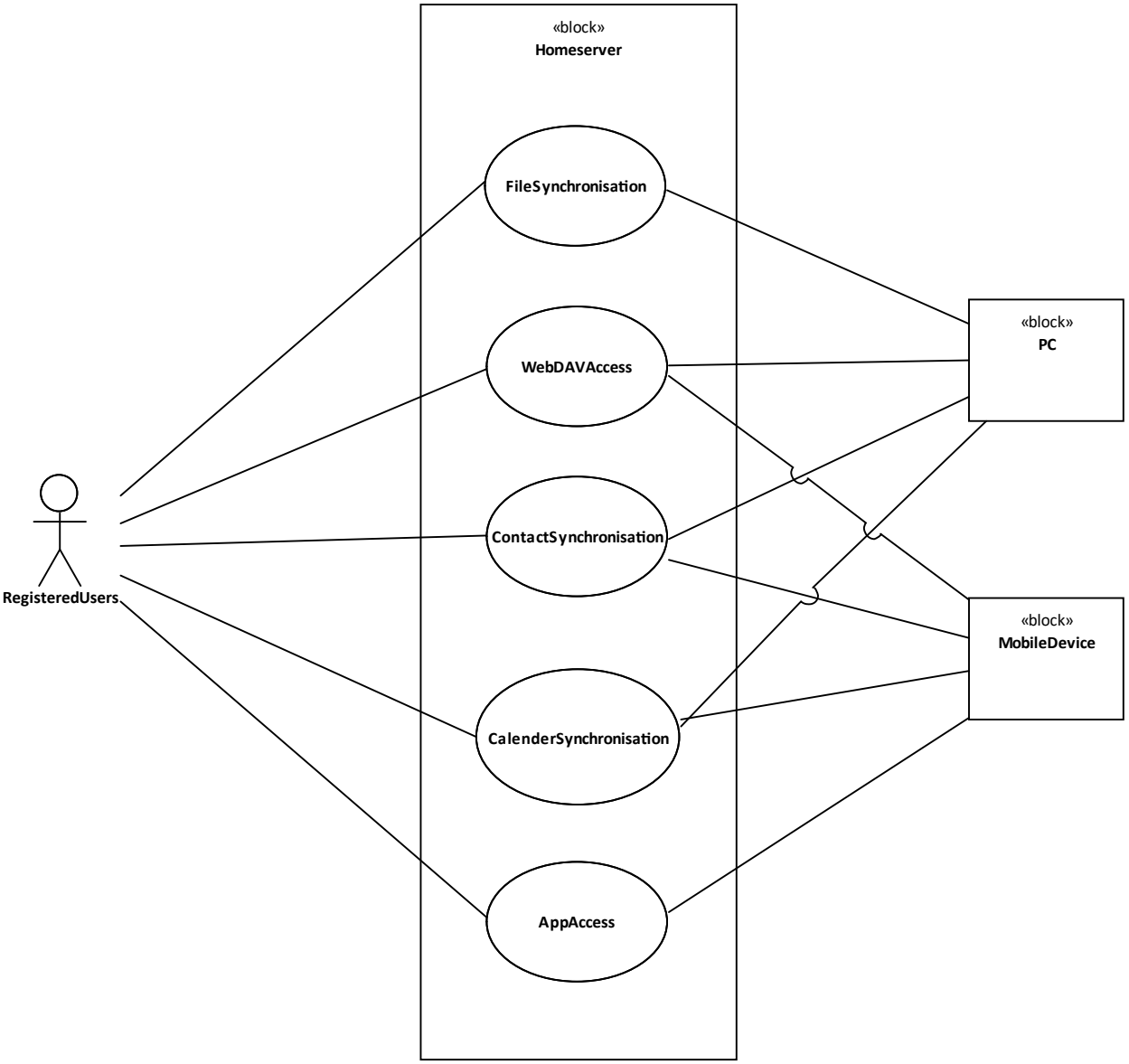

Abbildung 3.2: Verfeinerung des Anwendungsfalles „FileStorage"

4 Projektdurchführung

4.1 Hardware

Die Serverhardware umfasst folgende Komponenten:

- Fujitsu D3128-B25 Mainboard mit Intel C602 Server Chipsatz
- 8 x 8 GB DDR3-RDIMM ECC Arbeitsspeicher
- Intel XEON E5-2560V2 Octa-Core Server-CPU mit Hyperthreading
- Effizientes 450 W Modell Straight Power 11 von be quiet!
- Phanteks Enthoo Pro Tower PC-Gehäuse
- WD-Blue 250 GB SSD Festplatte
- ARCTIC Freezer 12 CO CPU-Kühler
- 4 x Seagate ST4000VN008 IronWolf 4 TB HDD Festplatte

All diesen Komponenten ist gemeinsam, dass sie auf Server-Anforderungen wie z.B. Dauerbetrieb optimiert sind. Des Weiteren sind viele der Komponenten gebraucht sehr günstig verfügbar.

4.2 Verwendete Basis-Software

4.2.1 Betriebssystem

Bevor ein sinnvolles Arbeiten auf dem Server möglich ist, steht die Entscheidung für ein bestimmtes Betriebssystem und dessen Installation an. Der Autor hat sich an dieser Stelle für den Einsatz eines UNIX-basierten Betriebssystems entschieden, da für solche Systeme die benötigte Software frei verfügbar ist. Als konkrete Distribution wurde Ubuntu Server 16.04 ausgewählt.

Die Installation des Betriebssystems gestaltet sich bei dem gewählten Mainboard etwas aufwändiger, weil dieses keine Integrierte Grafikschnittstelle bietet. An dieser Stelle ergeben sich zwei Möglichkeiten: Entweder es wird zeitweise eine Grafikkarte verwendet, welche über eine PCI-Express Schnittstelle angeschlossen werden kann, oder es wird eine *vollautomatische* Installation des OS durchgeführt. Im vorliegenden Projekt wurde die letztere Variante durchgeführt. Hierfür sind folgende Schritte notwendig:

1. **Erstellung einer Preseed-Datei:** Während der normalen interaktiven Installation werden dem Benutzer verschieden Fragen zur Konfiguration des Betriebssystems gestellt. Diese Fragen können *vor der Installation* in einer Datei beantwortet werden. Canonical stellt eine recht umfangreiche Dokumentation bereit, wie eine Preseed-Datei zu erstellen ist [8]. Die in diesem Projekt verwendete Preseeding-Datei stammt aus [1] und wurde an die Bedürfnisse angepasst.

2. **Bereitstellung der Datei:** Auch hier gibt es mehrere Möglichkeiten: Die Datei kann über einen FTP Server, auf dem Installationsdatenträger oder im initrd bereit gestellt werden. Die Bereitstellung über FTP bietet sich bei der Installation über PXE an. In diesem Fall erschien die einfachste Möglichkeit, die Datei auf dem Installationsmedium bereit zu stellen. Hierfür wird z.B. mit dem Programm **Cubic** ein angepasstes ISO-Image erstellt. Die Datei muss dann nur an eine beliebige Stelle in dem Image abgelegt werden, wobei sich der Ordner ,,preseed" anbietet.

3. **Bekanntmachung der Datei für den Bootloader:** Der Bootloader muss noch erfahren, dass er die Preseed-Datei nutzen soll, anstatt den Benutzer nach den Optionen zu fragen. Die folgenden Dateien stellen den Inhalt des Menüs dar, welches die Optionen beim starten der Installation entgegen nimmt. Der Option ,,Install Ubuntu Server" wird der Pfad der Preseed-Datei mit der Option `file=/cdrom/path/to/file` hinzu gefügt. Je nach BIOS-Version müssen unterschiedliche Dateien editiert werden.
   - Legacy: Die Datei `/isolinux/txt.cfg` muss editiert werden.
   - UEFI: die Datei `/boot/grub/grub.cfg` muss editiert werden.

4. **Weitere Optionen:** Es muss eine Standard-Option mit einem Timeout definiert werden, damit die Installation automatisch Startet. In der Datei `/isolinux/txt.cfg` geschieht das über den Eintrag `default <label>` und in der Datei `/boot/grub/grub.cfg` so wie in [5] beschrieben. Des Weiteren müssen noch die Optionen für `locale` z.B. `en_us` und `keyboard-configuration/layoutcode` z.B. mit `us` vorbelegt werden, da diese abgefragt werden, *bevor* der Bootloader die Preseed-Datei liest, wo diese Optionen ggf. auch spezifiziert wurden.

Es empfiehlt sich, die Installation vorher einmal in einer virtuellen Maschine zu testen, um festzustellen, ob diese dann auch vollautomatisiert durch läuft. Grundsätzlich sollte die Preseed-Methode auch für Ubuntu 18.04 funktionieren [8], jedoch hatte der Autor massive Probleme mit deren Ausführung und es wurde somit auf Ubuntu 16.04 zurück gegriffen. Ist es unabdingbar eine neuere Distribution zu nutzen, kann nach der Installation unkompliziert auf die neue Version upgegraded werden.

4.2.2 Containervirtualisierung mit Docker

Ist die Installation des Betriebssystems überstanden, eröffnen sich nun die Gestaltungsmöglichkeiten des eigentlichen Webservers. Dem Autor sind zwei grundsätzliche Herangehensweisen bekannt:

,,Harte" Installation der Komponenten auf der Maschine Es ist möglich, die gewünschte Software (z.B. einen Webserver, nextcloud, WordPress usw.) und die benötigten Abhängigkeiten (z.B. eine PHP-Laufzeitumgebung) einfach auf der Maschine zu installieren wie jedes andere Programm auch. Dies allerdings unschön im Bezug auf Flexibilität und Wartbarkeit: Nach einiger Zeit ist vergessen, welche Komponenten in welcher Version installiert wurden. Auch das Upgrade auf neue Versionen ist mit einem gewissen Risiko verbunden, da nicht mit garantiert werden kann, ob das System nach dem Upgrade noch so läuft wie es soll.

Aufteilung der Komponenten in Container Die bessere Lösung wäre eine standardisierte Laufzeit- und Entwicklungsumgebung, die sowohl auf dem Testsystem als auch auf

dem Produktivsystem identisch ist. Genau diese Funktionalität stellt Docker bereit: Eine Funktionalität wird in einem *Container* gekapselt. Ein Container ist vergleichbar mit einer virtuellen Maschine, die jedoch nativ im Kernel des Host-Systems läuft und dadurch wesentlich ressourcensparender ist als eine richtige VM. Alle benötigten Abhängigkeiten werden zur Laufzeit geladen und sind auch nur dann auf dem System vorhanden. Sobald der Container nicht mehr benötigt wird, kann dieser mit allen Abhängigkeiten mit einem Befehl vollständig entfernt werden.

Des Weiteren kann mit Docker risikofrei ein Versionsupgrade durchgeführt werden: Die neue Version einer Software wird lokal ausgiebig getestet und dann auf das Produktivsystem geladen. Durch einen Neustart der Container ist das Versionsupgrade vollzogen und es gibt nicht mal einen Moment Downtime.

Abgesehen von den Linux-Standard-Werkzeugen dem Dateisystem und einem Editor (der Autor nutzt GNU (x)Emacs) läuft sämtliche Software in Docker-Containern. Die Server-Software-Architektur ist in Kapitel 4.3 beschrieben.

4.2.3 Dateisystem

Als Dateisystem wird ZFS genutzt. Das von Sun entwickelte und von Oracle übernommenen Dateisystem stellt ein vollständiges Toolset zum Aufbau, zur Wartung und zur Wiederherstellung eines performanten Speicher-Backends zur Verfügung. Die Version ZFS-on-Linux ist darüber hinaus Open-Source und frei auf GitHub verfügbar. Zum Aufbau des hier verwendeten Speicher-Backends wurde als Ausgangspunkt das Tutorial zu ZFS von Aaron Toponce gewählt [7]. Im Folgenden wird kurz die Architektur des Speicher-Backends erläutert.

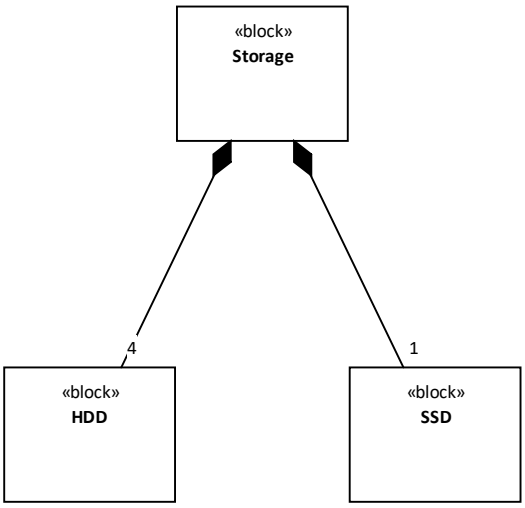

Abbildung 4.1: Der eingesetzte Massenspeicher besteht aus vier HDD Festplatten mit einer Kapazität von jeweils 4TB und einer SSD Festplatte mit einer Kapazität von 250GB, welche von ZFS als Schreib -und Lesecache genutzt wird.

Abbildung 4.1 zeigt die physische Zusammensetzung des Speicher-Backends. ZFS stellt sog. Z-Pools bereit. Diese sind i.d.R Zusammensetzungen aus verschiedenen physikalischen Festplatten. Die Organisation lautet wie folgt:

- Z-Pool `storage`:
  - RAIDZ1 (entspricht einem RAID5) bestehend aus
    - HDD1
    - HDD2
    - HDD3
  - ZFS Intent Log (ZIL, Schreibcache)
    - Partition 1 von der SSD
  - Adjustable Replacement Cache (ARC, Lesecache)
    - Partition 2 von der SSD
- Z-Pool `backup`
  - HDD4

Eine ähnliche Auflistung ist mit dem Kommando `zpool status` erzeugbar. ZFS zeigt schon hier seine Mächtigkeit, indem mit wenigen Befehlen ein RAID5 kombiniert mit einem schnellen Schreib-Lese-Cache aufgebaut werden kann. Des Weiteren ist das dargestellte System immun gegen Inkonsistenzen, da es sich bei ZFS um ein Copy-on-Write-Dateisystem handelt, das bedeutet, dass bei der Änderung einer Datei diese erst kopiert dann geändert und dann ersetzt wird. Wird eine Änderung unplanmäßig unterbrochen, so ist diese höchsten verloren, jedoch die Datei immer noch konsistent.

4.2.4 Automatisiertes Backup

Für das automatisierte Backup wird das Feature Snapshots kombiniert mit den Tools ZFS-Auto-Snapshot [9] sowie Syncoid [6] eingesetzt. Ein Snapshot ist eine Momentaufnahme des Standes eines Z-Pools vergleichbar mit einem git commit. Das Tool ZFS-Auto-Snapshot produziert und löscht periodisch Snapshots welche mit dem Tool Syncoid einmal täglich vom Produktiv-Pool auf den Backup-Pool synchronisiert werden. Die Zeitsteuerung erfolgt über Cron-Jobs angelehnt an die Angaben aus [2].

4.3 Softwarearchitektur

4.3.1 Deployment der Container und der Anwendungen

Im Folgenden wird das Deployment der Anwendungen auf dem Server und die interne Kommunikation der Anwendungen vorgestellt

Wie bereits in 4.2 angedeutet, werden die Applikationen durch Kapselung in Docker-Containern organisiert und vom Betriebssystem getrennt.

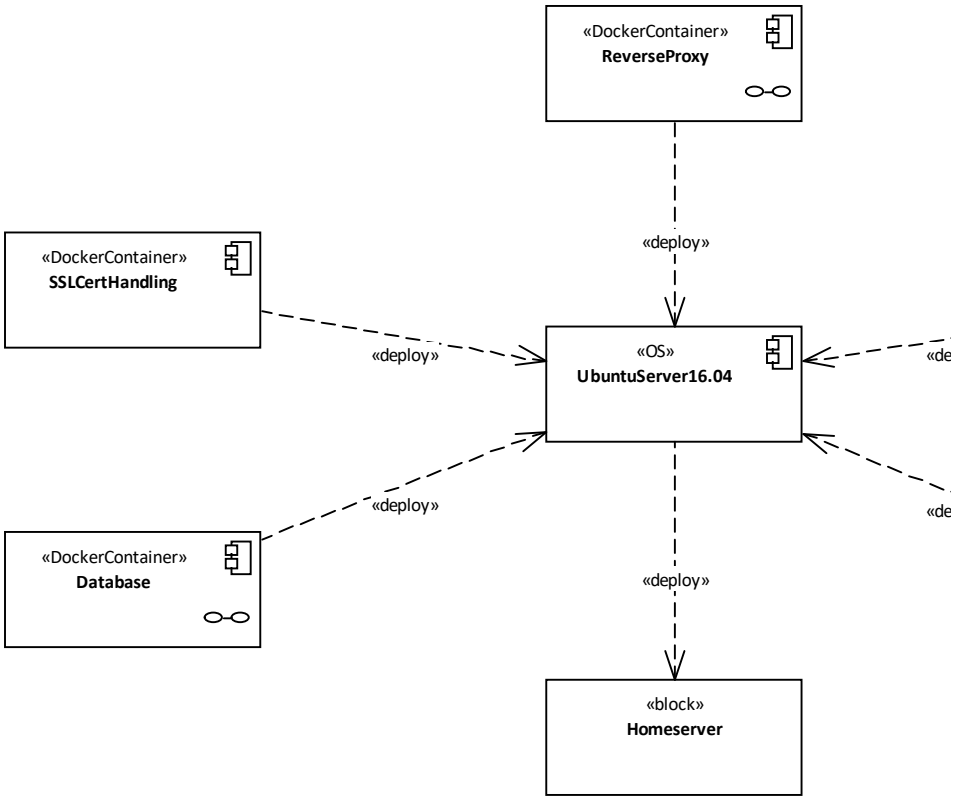

Abbildung 4.2: Komponentendiagramm der eingesetzten Softwarekomponenten und ihre Beziehung zum Betriebssystem. Die Abhängigkeit stereotypisiert mit ≪deploy≫ bedeutet hierbei, dass die Komponente die jeweilige Zielkomponente als Laufzeitumgebung nutzt.

Das Deployment der Anwendungen selbst erfolgt dann auf die Docker-Container, wie in Abbildung 4.3 gezeigt. Je nach Konfiguration findet die Anwendung ein vollständiges Betriebssystem mit allen Abhängigkeiten vor, die im Dockerfile definiert wurden.

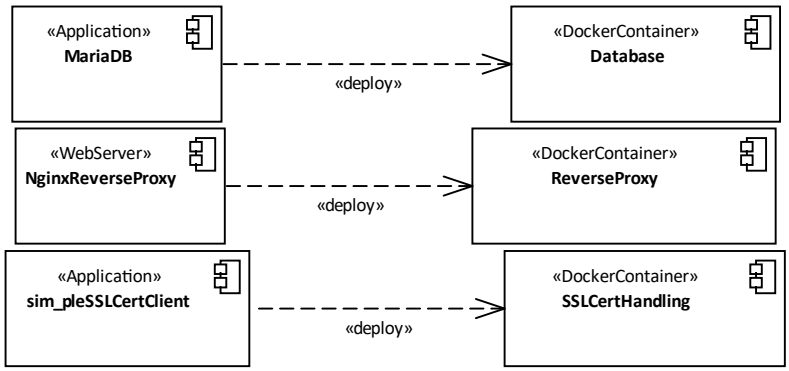

Abbildung 4.3: Deployment der Anwendungen auf die dazugehörigen Docker-Container

4.3.2 Starten der Docker-Container

Die einfachste Möglichkeit, einen Docker-Container zu starten, ist die Eingabe des Befehls `docker run <image_name>`. Ein Image ist ein mit `docker build` gebautes Dockerfile vergleichbar mit dem Image eines Betriebssystems. Mit dem `docker run` Befehl wird dieses gestartet und es entsteht die Instanz eines Images, ein Container. Mehr Informationen zum Befehl `docker run` ist in [4] zu finden.

Die beschriebene Methode ist für einfache Tests gut verwendbar. Ist jedoch ein komplexes Gebilde aus Containern zu bauen, so ist es nicht zielführend bei jedem Start alle Container von Hand zu starten. Hierfür wird von Docker das Werkzeug Docker Compose bereit gestellt. Dieses liest eine Datei aus (standardmäßig `docker-compose.yml`), in der alle Optionen definiert sind, mit denen die Container sonst mit `docker run` gestartet werden. Mehr Informationen zu Docker Compose sind in [3] zu finden.

Im hier vorgestellten System wurde eine `docker-compose.yml`-Datei für Basisdienste wie die Datenbank und den Reverse Proxy, sowie eine Datei für die konkrete Anwendung wie die Nextcloud. Dieses Verfahren ermöglicht eine getrennte Verwaltung von Infrastruktur und Anwendung.

4.3.3 Interne Kommunikation der Container

Um eine Kommunikation zwischen den Containern zu ermöglichen wird das Feature *network* von Docker genutzt. Dieses ermöglicht den Aufbau virtueller TCP/IP-Netzwerke innerhalb einer Maschine (=*Bridge*-Netzwerk) oder auch maschinenübergreifend (=*Overlay*-Netzwerk) zum Aufbau von Clustern. Da jeder Container eine kleine virtuelle Maschine darstellt, verfügt auch jeder Container über eine Netzwerkschnittstelle, mit der er dem ausgewählten Netzwerk beitreten kann.

Im vorliegenden Projekt wurden ausschließlich Bridge Netzwerke genutzt, da nur eine Maschine vorhanden ist. Dabei wurden die Netzwerke *frontend* und *backend* eingerichtet. Dies ermöglicht eine virtuelle Trennung zwischen der Benutzerseite (z.B. der Reverse-Proxy, der Nutzeranfragen von außen verarbeitet) und dem Server-Backend (z.B. die Datenbank). Abbildung 4.4 zeigt die Beziehung der Container untereinander.

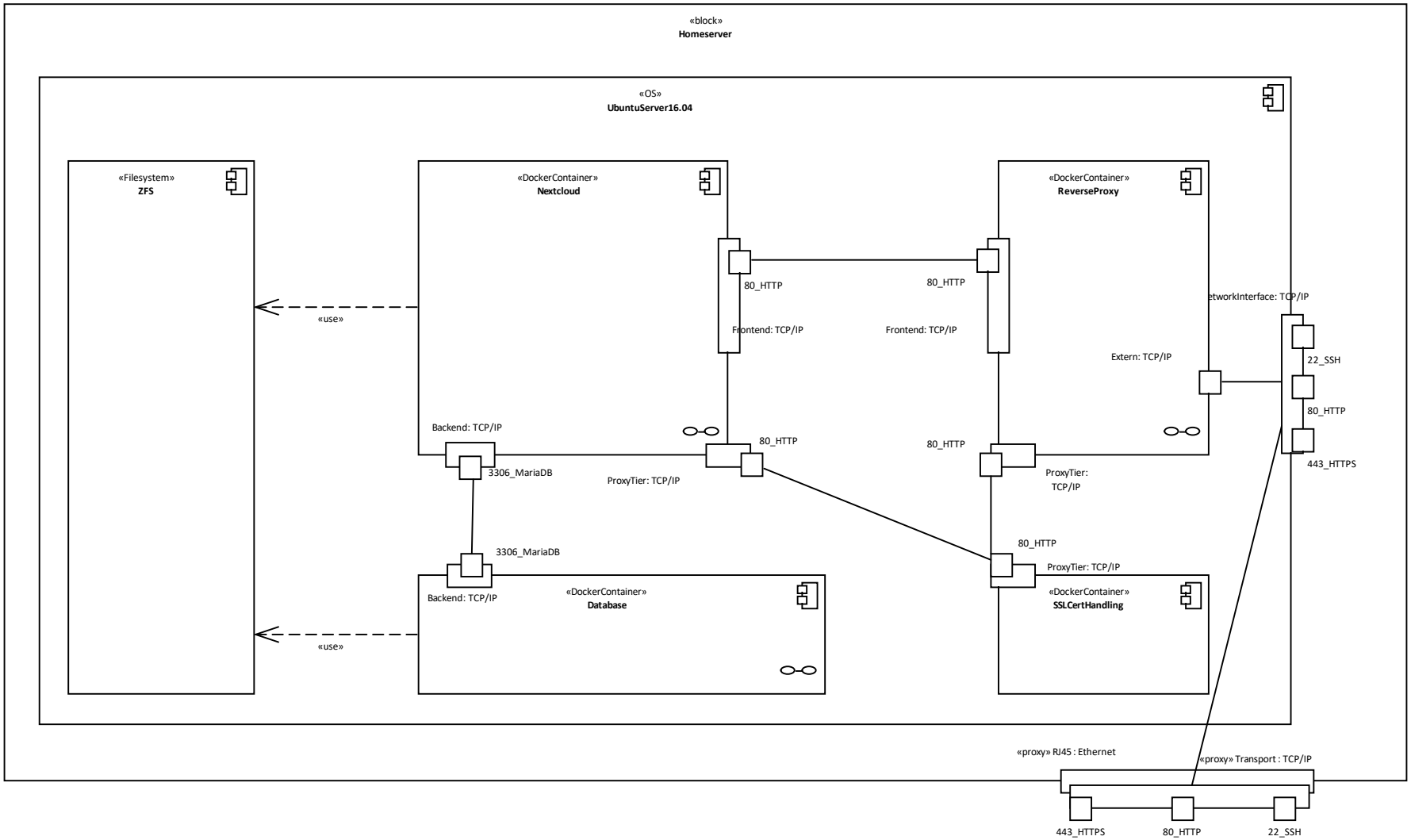

Abbildung 4.4: Kompositionsdiagramm der eingesetzten Docker-Container. Die Docker-Container sind jeweils über die passenden Bridge-Netzwerke `frontend` und `backend` miteinander verbunden. Die Datenbank und die Nextcloud nutzen den von ZFS verwalteten Speicherpool als Speicher-Backend

4.4 Webdesign mit Django

Eine weiterer Anwendungsfall für einen Homeserver ist das Betreiben eines eigenen Webauftrittes. Zunächst war geplant, diesen mit WordPress zu gestalten. Eine WordPress-Instanz laufen zu lassen ist ähnlich unkompliziert wie das starten der Nextcloud wie in Listing A.2 in Anhang A beschrieben. Dies wurde auch gemacht und es wurde eine Weile mit WordPress experimentiert jedoch ergaben sich dabei folgende Erkenntnisse:

- Pro
  - WordPress ist sehr einfach in Betrieb zu nehmen und es geht schnell einen ersten Webauftritt zu gestalten
  - WordPress ist im Prinzip gut erweiterbar mit eigenen Themes
  - Es gibt eine ganze Reihe fertig verfügbarer Plugins und Themes die nur geladen werden müssen und dann sofort genutzt werden können.
- Kontra
  - WordPress ist in PHP geschrieben, eine Sprache in der alles möglich ist, in der jedoch auch alles selber gemacht werden muss. Für Anfänger birgt diese Sprache ähnlich wie C das Risiko schwere Sicherheitslücken zu produzieren.
  - WordPress ist sehr auf den Anwendungsfall Webblog ausgerichtet. Andere Anwendungen sind möglich, jedoch ist die WordPress API strickt vorgegeben.

Literatur

[1] *Beschreibung der automatischen Installation von theurbanpenguin.*
URL: `https://www.theurbanpenguin.com/auto-installing-ubuntu-16-04/` (besucht am 25. 10. 2019).

[2] *Cron Job ZFS Synchronisation.*
URL: `https://serverfault.com/questions/842531/how-to-perform-incremental-continuous-backups-of-zfs-pool` (besucht am 12. 01. 2020).

[3] *Docker Reference - Compose.*
URL: `https://docs.docker.com/compose/` (besucht am 12. 01. 2020).

[4] *Docker Reference - Run.*
URL: `https://docs.docker.com/engine/reference/run/` (besucht am 12. 01. 2020).

[5] *GRUB2 Konfiguration - ubuntuusers.de.*
URL: `https://wiki.ubuntuusers.de/GRUB_2/Konfiguration/` (besucht am 25. 10. 2019).

[6] *Sanoid und Syncoid.*
URL: `https://github.com/jimsalterjrs/sanoid` (besucht am 12. 01. 2020).

[7] Aaron Toponce.
ZFS Tutorial.
URL: `https://pthree.org/2012/04/17/install-zfs-on-debian-gnulinux/` (besucht am 12. 01. 2020).

[8] *Ubuntu Manual - Installation mittels Preseeding.*
URL: `https://help.ubuntu.com/lts/installation-guide/amd64/apb.html` (besucht am 25. 10. 2019).

[9] *ZFS Auto Snapshot.*
URL: `https://github.com/zfsonlinux/zfs-auto-snapshot` (besucht am 12. 01. 2020).

A Docker-Compose Dateien

```
version: '3.1'

services:
  nginx-proxy:
    build: ./proxy
    container_name: nginx-proxy
    ports:
      - 80:80
      - 443:443
    volumes:
      - certs:/etc/nginx/certs:ro
      - vhost.d:/etc/nginx/vhost.d
      - html:/usr/share/nginx/html
      - /var/run/docker.sock:/tmp/docker.sock:ro
    networks:
      - frontend
      - proxy-tier

  letsencrypt-companion:
    image: jrcs/letsencrypt-nginx-proxy-companion
    container_name: nginx-proxy-le
    restart: always
    volumes:
      - certs:/etc/nginx/certs:rw
      - vhost.d:/etc/nginx/vhost.d
      - html:/usr/share/nginx/html
      - /var/run/docker.sock:/var/run/docker.sock:ro
    environment:
#     -
↪   ACME_CA_URI=https://acme-staging-v02.api.letsencrypt.org/directory
      - NGINX_PROXY_CONTAINER=nginx-proxy
    networks:
      - frontend
      - proxy-tier

  db:
    image: mariadb
    container_name: db
    command: --transaction-isolation=READ-COMMITTED --binlog-format=ROW
    restart: always
    volumes:
```

```
      - /home/fuchs/storage/mariadb:/var/lib/mysql
      - ./docker-entrypoint-initdb.d:/docker-entrypoint-initdb.d
    environment:
      MYSQL_ROOT_PASSWORD: WOLF_FUCHS_20052018_db
    networks:
      - backend

#  adminer:
#    image: adminer
#    restart: always
#    ports:
#      - 8080:8080
#    networks:
#      - backend

volumes:
  certs:
  vhost.d:
  html:

networks:
  backend:
    external:
      name: backend
  frontend:
    external:
      name: frontend
  proxy-tier:
    external:
      name: proxy-tier
```

Listing A.1: Docker-compose file für die Datenbank, den Reverse-Proxy und die SSL-Zertifikatsverwaltung

```
version: '3.1'

services:

  nextcloud:
    image: nextcloud
    restart: always
    ports:
      - 80
    volumes:
      - /path/to/storage/nextcloud/html:/var/www/html
      - /path/to/storage/nextcloud/custom_apps:/var/www/html/custom_apps
      - /path/to/storage/nextcloud/config:/var/www/html/config
      - /path/to/storage/nextcloud/data:/var/www/html/data
    environment:
      MYSQL_HOST: db
      MYSQL_PASSWORD: <save_pw>
      MYSQL_DATABASE: db_nextcloud
      MYSQL_USER: nextcloud
      VIRTUAL_HOST: <domain_der_nextcloud>
      LETSENCRYPT_HOST: <domain_der_nextcloud>
      LETSENCRYPT_EMAIL: <erreichbare_mail_adresse>
      NEXTCLOUD_ADMIN_USER: <administrator_name>
      NEXTCLOUD_ADMIN_PASSWORD: <save_pw>
      NEXTCLOUD_TRUSTED_DOMAINS: <domain_der_nextcloud>
    networks:
      - frontend
      - backend
      - proxy-tier

networks:
  frontend:
    external:
      name: frontend
  backend:
    external:
      name: backend
  proxy-tier:
    external:
      name: proxy-tier
```

Listing A.2: Docker-compose file für die Nextcloud