Abschlussprüfung

Fachinformatiker für Systemintegration

Dokumentation zur betrieblichen Projektarbeit

b2Xproc Deployment

Umsetzung einer automatisierten Lösung zur Bereitstellung personalisierbarer B2B Webshops mit gesichertem Zugriff aus dem Internet

Bearbeitungszeitraum:

2. Halbjahr 2024/2025

24.02.2025 - 19.04.2025

Bearbeiter:

Heinrich Thaler

www.it-thaler.de

info@it-thaler.de

Praktikumsbetrieb:

ANTHOLZER GmbH & Co. KG

www.antholzer.de

info@antholzer.de

Inhaltsverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bezeichnung
API	Application Programming Interface
ORM	Object-Relational Mapping
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UAT	User Acceptance Tests
URL	Uniform Resource Locator
VM	Virtual Machine
WSGI	Python Web Server Gateway Interface

1. Einleitung

1.1. Projektziel

Abbildung 1: Logo von b2Xproc

Ziel des Projektes ist die automatisierte Bereitstellung neuer Webshop-Instanzen von b2Xproc. Über eine Weboberfläche, dem sogenannten Shop-Manager, erhalten autorisierte Mitarbeiter der Firma Antholzer die Möglichkeit, eigenständig und ohne Unterstützung weitere Shops zu erstellen. Dabei sollen die Webshops sicher und zuverlässig aus dem Internet erreichbar sein. Die Bereitstellung der Shops soll vollständig automatisiert erfolgen und einen minimalen manuellen Aufwand erfordern. So wird es Kunden ermöglicht, bereits während eines Termins vor Ort einen Prototyp ihres personalisierten Webshops live einzusehen.

1.2. Projektbegründung

Potenzielle Kunden, also Firmen, welche einen personalisierten Shop benötigen, möchten oft vor dem Kauf einen Prototyp sehen. Bisher war dieser Vorgang mit erheblichem Aufwand für die IT-Abteilung verbunden, da die Erstellung neuer Shops manuell erfolgen musste. Dadurch konnte es bis zu zwei Wochen dauern, bis die IT Kapazität hatte, neue Shops einzurichten.

Durch die Verwendung des Shop-Managers entfällt diese Wartezeit. Mitarbeiter können direkt über die Benutzeroberfläche Shops erstellen, ohne technische Kenntnisse oder IT-Support zu benötigen. Dies sorgt für eine deutlich schnellere Reaktionszeit und verbessert die Effizienz im Vertrieb.

1.3. Projektumfeld

Das Projekt wird an den EDV-Schulen des Landkreises Deggendorf im Rahmen der Ausbildung zum Fachinformatiker für Systemintegration umgesetzt. An der Berufsfachschule erfolgt die Ausbildung praxisnah, mit einem starken Fokus auf die Anwendung der erworbenen Kenntnisse im späteren echten Berufsleben.

Die konkrete Projektumsetzung erfolgt als Praktikumsarbeit im Rahmen eines halbjährigen Praktikums im Anschluss an die theoretische Ausbildung. Praxisbetrieb ist die Firma ANTHOLZER.

Die Firma ANTHOLZER GmbH & Co. KG ist ein Komplett-Anbieter für Arbeits- und Teamkleidung, inklusive eigener In-House Produktion für die Veredelung mittels Stick und Druck. Zudem bieten sie ihren Kunden im Bereich E-Commerce Corporate-Webshops an.

b2Xproc ist das neue Shopsystem, das von der Firma ANTHOLZER In-House entwickelt wird. Der Shop an sich ist nicht Teil dieses Projektes, sondern nur seine Integrierung in den Shop-Manager.

2. Projektplanung

2.1. Projektphasen

Diese Tabelle enthält eine grobe Zeitplanung zu den verschiedenen Projektphasen:

Projektphase	Geplante Zeit
Einarbeitung	3 Stunden
IST-Analyse	2 Stunden
SOLL-Analyse	3 Stunden
Einrichtung Docker mit Traefik	5 Stunden
Dockerfile und Image erstellen	3 Stunden
Skripte für Hetzner API	2 Stunden
Webshoperstellung per Skripte	5 Stunden
Bedienung per Weboberfläche	6 Stunden
Testphase	6 Stunden
Erstellung Dokumentation	5 Stunden
Gesamt	**40 Stunden**

2.2. Ressourcenplanung

Dieses Projekt besteht ausschließlich aus Open-Source-Software. Für die Shops wird eine Domain benötigt, die bei Hetzner erworben wird. Die kostenlose DNS-Console ermöglicht die Erstellung von Subdomains über eine API. Zudem wurde als Testumgebung eine VM in der Cloud benutzt, welche aus dem Internet erreichbar ist.

Für die Entwicklung mit Python wurde PyCharm Community Edition verwendet. Der Zugriff auf den Cloud-Server erfolgt über SSH.

Während der Entwicklung stehen ein Büro, ein PC sowie die nötige Infrastruktur zur Verfügung

2.3. Entwicklungsprozess

Da dieses Projekt eigenständig durchgeführt wird, kommt das Wasserfallmodell als Entwicklungsmethode zum Einsatz. Dieses Modell ist besonders für Einzelprojekte geeignet, da es einen klar strukturierten Ablauf bietet. Die in 2.1 beschriebenen Phasen werden nacheinander abgearbeitet.

3. Analysephase

3.1. Ist-Analyse

Ein Großteil der Firmenkunden bestellen für ihre Mitarbeiter regelmäßig Teamwear, Workwear oder Schutzausrüstung mit dem eigenen Firmenlogo. Meist erfolgt dies umständlich mittels Excel-Tabellen. Um den Beschaffungsprozess kundenseitig nachhaltig zu erleichtern, werden für Großkunden eigene personalisierte Webshops angeboten, in denen Textilien mit Veredelung bestellt werden können. In den Shops kaufen die Mitarbeiter überwiegend selbst ein. Ihnen steht bspw. ein individuell festgelegtes Budget für die Bestellung der Kleidung zur Verfügung.

Wenn ein Kunde sich für einen eigenen Webshop entschieden hat, dauert es ab der Freigabe aktuell mindestens zwei Wochen bis dieser bereitgestellt werden kann. Dabei müssen folgende Schritte erledigt werden, welche auf einen kombinierten Arbeitsaufwand von 8 bis 10 Stunden hinauslaufen: Shoperstellung und -design, Einpflegen von Artikel- und Kundenstammdaten sowie Konfiguration der benötigten Funktion. Die IT-Abteilung ist für die Erstellung der Webshop-Instanzen zuständig und benötigt dafür zwei Stunden.

Zurzeit wird ein neuer Webshop in Python mit Django entwickelt. Mit diesem neuen Webshop-System werden alle zuvor beschriebenen Schritte verkürzt – bis auf die Erstellung einer Webshop-Instanz. An dieser Stelle setzt dieses Projekt an, indem es den ersten Einrichtungsschritt, d. h. die Erstellung der Instanz, verkürzt. Eine Automatisierung dieses Schrittes ist die Basis für die Verkürzung des gesamten Arbeitsaufwandes auf eine Stunde und 30 Minuten.

Es sind zwei Server innerhalb des Firmengebäudes vorhanden, jedoch sind die Webshops auf einer dedizierten Maschine in der Cloud, gehostet von Hetzner.

3.2. Wirtschaftlichkeitsanalyse

3.2.1. Projektkosten

Einmalige Kosten

	Berechnung	Kosten
Entwicklungskosten	40 h * 60 €	2.400,00 €
Lizenzkosten	Ausschließlich Open-Source-Software	0,00 €
Gesamt		**2.400,00 €**

Jährliche Kosten

	Berechnung	Jährliche Kosten
Domain		11,90 €
VM	6,30 € / Monat * 12 Monate	75,60 €
Gesamt Jährlich		**87,50 €**

3.2.2. *Amortisationsdauer*

Jährliche Einsparung

Im Schnitt wird pro Woche ein neuer Shop beantragt. Die IT-Abteilung benötigt für die Bearbeitung eines Shops 2 Stunden. Der Shop-Manager eliminiert den Arbeitsaufwand für die IT-Abteilung.

	Berechnung	Jährliche Einsparung
Zeiteinsparung	1 Shop / Woche → 2 h/Woche gespart 2h * 4 Wochen * 12 Monate = 96 h/Jahr 96 h * 60 €	5.760,00 €
Gesamt Jährlich		**5.760,00 €**

Amortisationszeit:	Einmalige Kosten / (Jährliche Einsparung – Jährliche Kosten)	0,42 Jahre

0,42 Jahre * 12 Monate ≈ 5 Monate

Die Kosten des Projektes sind nach 5 Monaten durch die jährlichen Einsparungen vollständig amortisiert. Die nachfolgende Abbildung veranschaulicht den Vergleich der monatlichen Kosten vor und nach der Projektumsetzung:

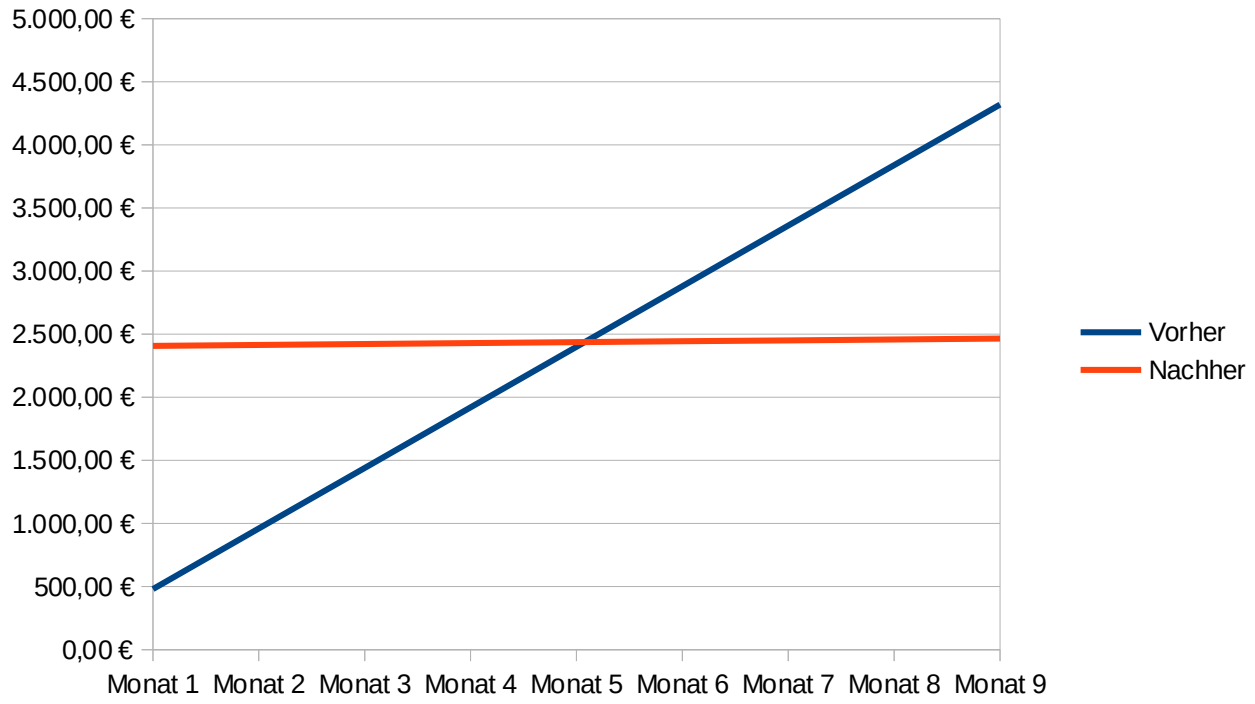

Abbildung 2: Amortisationsdauer

3.2.3. Qualitativer Nutzen

Durch die Automatisierung des Webshop-Erstellungsprozesses entfällt die bisherige Reaktionszeit von zwei Wochen, da kein manueller Eingriff der IT-Abteilung erforderlich ist. Die Mitarbeiter der Abteilung Backoffice können bei Bedarf sofort über den Shop-Manager einen neuen Shop erstellen, beispielsweise um kurzfristig einen Test-Shop bei einem Kundentermin zu erstellen. Außerdem führt diese Automatisierung zu einer erheblich geringeren Fehleranfälligkeit, da manuelle Arbeitsschritte reduziert werden. Gleichzeitig gewährleistet das System eine hohe Skalierbarkeit, sodass mit dem weiteren Wachstum des Unternehmens jederzeit zusätzliche Webshops ohne Mehraufwand eingerichtet werden können. Insgesamt entlastet dieser Ansatz die IT-Abteilung nachhaltig.

3.3. Qualitätsanforderungen

3.3.1. Einfache & Schnelle Bedienung

Da diese Anwendung primär vom Backoffice verwendet wird, muss diese einfach und ohne Fachkenntnis zu bedienen sein. Der Prozess um einen Shop zu erstellen muss möglichst schnell ablaufen, damit diese bei Bedarf auch vor Ort bei potenziellen Kunden erstellt werden können.

3.3.2. Wartbarkeit

Um eine langfristige Pflege und Weiterentwicklung zu gewährleisten, muss der Code leicht verständlich, sauber strukturiert und gut dokumentiert sein. Dies erleichtert nicht nur zukünftige Anpassungen, sondern reduziert auch den Einarbeitungsaufwand für neue Entwickler.

Da APIs und andere externe Dienste sich in der Zukunft ändern oder durch Alternativen ersetzt werden könnten, sollte das System so konzipiert sein, dass ein Austausch dieser Schnittstellen mit minimalem Aufwand möglich ist.

3.3.3. Sicherheit

Da die Webshops sensitive Daten wie Zahlungsinformationen, Kunden- und Bestelldaten übertragen, ist eine SSL-Verschlüsselung zwingend erforderlich. Durch den Einsatz von TLS (Transport Layer Security) wird sichergestellt, dass alle Daten verschlüsselt zwischen dem Client und dem Server übertragen werden, wodurch sie vor Man-in-the-Middle-Angriffen und anderen Sicherheitsrisiken geschützt sind.

Die SSL-Zertifikate müssen über eine vertrauenswürdige Zertifizierungsstelle ausgestellt und regelmäßig erneuert werden. Zudem sollen alle Verbindungen ausschließlich über HTTPS erfolgen, indem HTTP-Anfragen automatisch auf HTTPS umgeleitet werden.

4. Entwurfsphase

4.1. Zielplattform

Da bei weiterem Wachstum ein Umzug auf Kubernetes geplant ist, wird das gesamte Projekt in Docker-Containern umgesetzt. Dank Docker ist das System unabhängig von Betriebssystem und Hardware.

Der Shop-Manager wird in Python geschrieben, da es eine schnelle Entwicklung ermöglicht. Zudem ist es die meistgenutzte Programmiersprache bei der Firma Antholzer.

Für das Frontend wird Flask verwendet. Zunächst, wie im Projektantrag beschrieben, war Django vorgesehen. Django und Flask haben sehr viele Gemeinsamkeiten: Beide benutzen Python und haben ähnliche Template-Sprachen. Der Hauptunterschied zwischen den beiden Frameworks liegt in der Anzahl der Features. Django bringt viele Tools mit, darunter ORM (Object-Relational Mapping), Adminpanel und Authentifikation. Bei Flask müssen solche Tools über externe Pakete installiert werden. Da Flask dadurch leichtgewichtiger ist und besonders für kleinere Projekte, wie dieses, schnellere Ergebnisse erzielt werden können, habe ich mich dafür entschieden. Django ist für große Projekte besser geeignet, wie z.B. das Shopsystem b2xproc an sich.

Für eine möglichst hohe Flexibilität wurde MongoDB als Datenbank gewählt.

4.2. Projektaufbau

Im folgenden Diagramm wird der Aufbau des Projektes dargestellt:

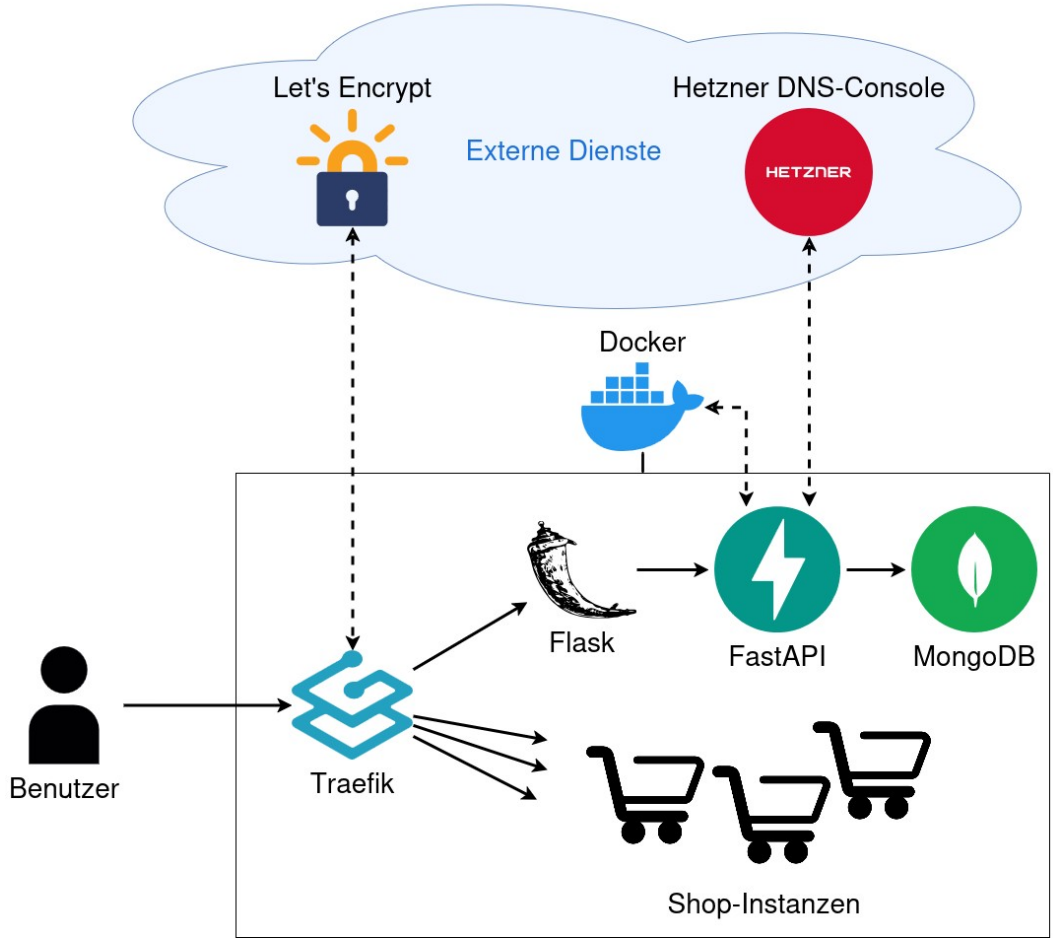

Abbildung 3: Projektstruktur

Alle Dienste innerhalb des Rechtecks sind Docker-Container.

- Docker: Runtime der Container. Alle Teile dieses Projektes wurden in Docker-Containern umgesetzt.
- Traefik: Dient als Reverse Proxy und Loadbalancer. Automatische SSL-Verschlüsselung mit Zertifikaten von Let’s Encrypt.

- Shop-Instanzen: Mehrere voneinander unabhängige Instanzen des neuen Shop-Systems. Eine Instanz besteht aus mehreren Containern.
- FastAPI: Das Backend zur Verwaltung der Shops. Greift dafür direkt auf den Docker-Host zu.
- Flask: Das Frontend für die Bedienung des Shop-Managers. Kümmert sich um die Authentifizierung.

Die Benutzer des gesamten Systems lassen sich in zwei Gruppen unterteilen: Zum einen Kunden, die die Webshops nutzen, und zum anderen Mitarbeiter, die den Shop-Manager benutzen.

Die vollen Pfeile stellen den Verlauf der Anfragen dar. Traefik entscheidet anhand der Domain und der URL an welchen Container die Anfragen weitergeleitet werden sollen.

Die gestrichelten Linien stellen Schnittstellen dar: Traefik holt SSL-Zertifikate von Let’s Encrypt, die API registriert Subdomains bei Hetzner und steuert den Docker Host.

4.3. Anwendungsfälle

Es gibt zwei Gruppen, welche den Shop-Manager verwenden werden: das Backoffice und die IT-Abteilung (Abbildung 4). Das Backoffice soll nur Shops erstellen dürfen, während die IT auch ganze Shops löschen darf. Auch die Benutzerverwaltung soll nur von den Administratoren verwendet werden können.

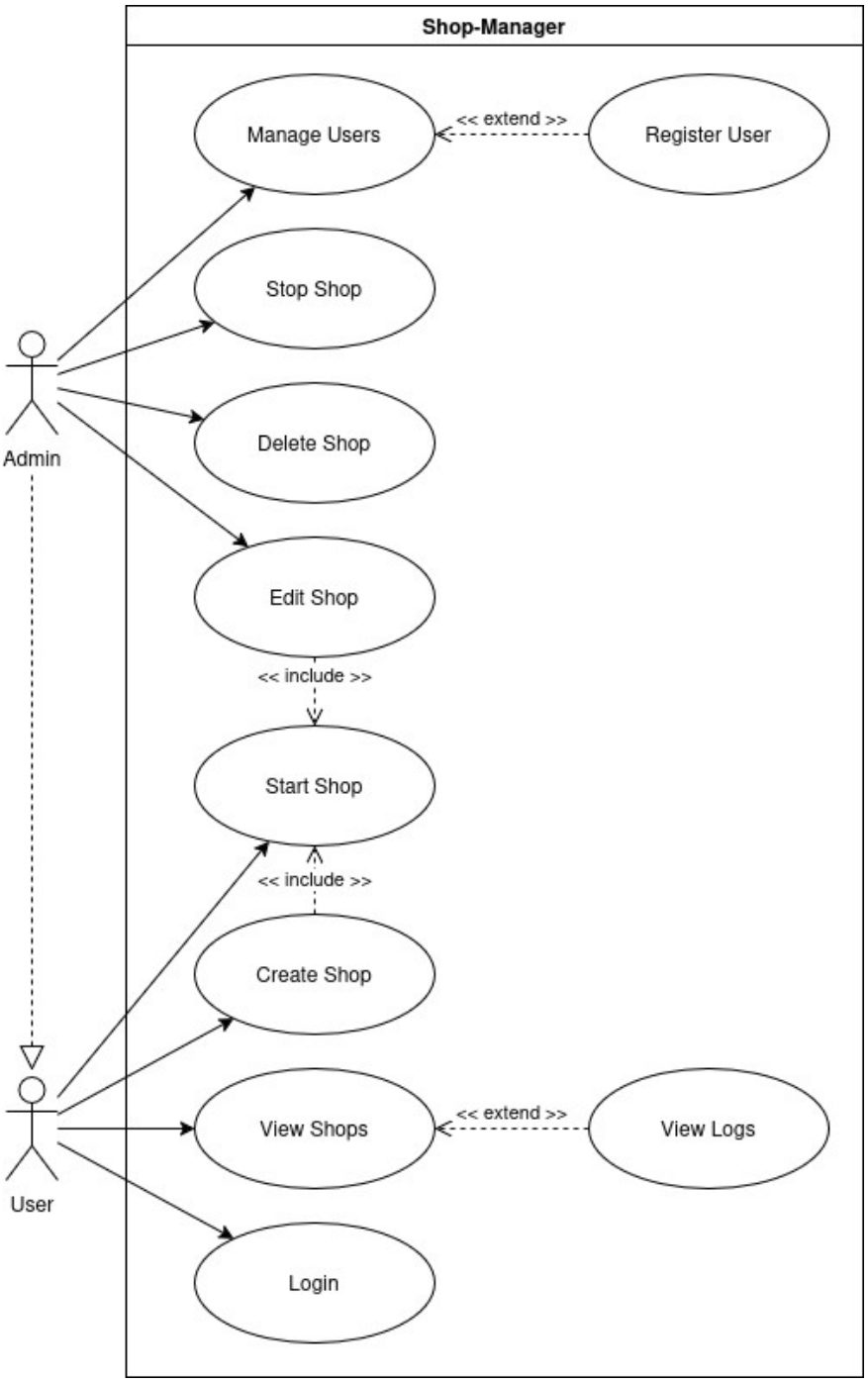

Abbildung 4: Use-Case Diagramm des Shop-Managers

5. Implementierungsphase

5.1. Einrichtung Docker & Traefik

Die Installation von Docker verlief reibungslos und war dank der gut strukturierten offiziellen Anleitung [1] innerhalb weniger Minuten abgeschlossen.

Traefik wurde mithilfe von Docker-Compose installiert. Anschließend musste es mit einer YAML-Datei konfiguriert werden (Anhang 9.2):

```
[...]
22  providers:
23    docker:
24      exposedByDefault: false
25      network: proxy
[...]
31  entryPoints:
32    web:
33      address: ":80"
34      http:
35        redirections:
36          entryPoint:
37            to: websecure
38            scheme: https
39    websecure:
40      address: ":443"
41      asDefault: true
42      http:
43        tls:
44          certResolver: letsencrypt
[...]
48  certificatesResolvers:
49    letsencrypt:
50      acme:
51        email: heinrich.thaler@antholzer.de
52        storage: "/letsencrypt/acme.json"
53        tlsChallenge: {}
```

Container aus dem „proxy“ Netzwerk mit einem speziellen Label werden automatisch freigegeben.

Alle HTTP Anfragen werden auf HTTPS umgeleitet.

HTTPS auf Port 443 verwendet automatisch TLS.

Konfiguration von Let’s Encrypt für die Bereitstellung von SSL-Zertifikaten.

In Zeile 22 werden die „providers“ definiert. Der Eintrag „docker“ bewirkt, dass Container mit bestimmten Labels automatisch von Traefik erkannt und über das Internet erreichbar werden:

```
labels:
  - "traefik.enable=true"
  - "traefik.http.routers.shop-manager-ui.rule=Host(`antholzer-shop.de`) && PathPrefix(`/shop-manager`)"
```

Dieser Ausschnitt aus einer Docker Compose Datei stellt einen Container unter https://antholzer-shop.de/shop-manager zur Verfügung.

5.2. Shop dockerisieren

Parallel zu diesem Projekt wird zurzeit ein neues Shop-System (b2Xproc) entwickelt. Dieses soll als Docker-Container laufen. Dafür musste ich ein Dockerfile erstellen, welches alle benötigten Bibliotheken mit pip installiert. Beim Start des Containers wird dieses Entrypoint-Script ausgeführt:

```
1  #!/bin/bash
2  set -eo pipefail # Exit script if any commands fail
3
4  echo "=== Django Entrypoint Script V1 - Heinrich Thaler"
5  echo "--- Collecting Static Files..."
6  python manage.py collectstatic --noinput
7
8  echo "--- Making migrations..."
9  python manage.py makemigrations
10
11 echo "--- Migrating..."
12 python manage.py migrate
13
14 CONTAINER_FIRST_STARTUP="/data/CONTAINER_STARTED_BEFORE_1"
15 if [ ! -e $CONTAINER_FIRST_STARTUP ]; then
16     echo ">>> Detected first startup of Container!"
17
18     echo "--- Creating superuser"
19     python manage.py createsuperuser --noinput --username admin --email admin@admin
20
21     echo "--- Populating Initial Demo Data"
22     python manage.py init_data --demo
23
24     touch $CONTAINER_FIRST_STARTUP
25     echo "<<<"
26 fi
27
28 unset DJANGO_SUPERUSER_PASSWORD
29
30 echo "--- Starting application..."
31 echo "=== Entrypoint Script Done!"
32 echo
33 gunicorn --bind 0.0.0.0:8000 --workers 3 b2xproc.wsgi:application
```

Bei einem Fehler wird die Ausführung des Scripts sofort abgebrochen.

Initialisierung der Datenbank.

Block wird nur bei erstem Start ausgeführt.

Administrator-Benutzer erstellen.

Datenbank mit initialen Daten befüllen.

Startet den Shop.

Anschließend wurde das Image gebaut und ein Test-Shop mit Docker-Compose aufgesetzt. In der Compose-Datei (Anhang 9.3) wurden drei Services definiert:

1. db: PostgreSQL Datenbank in der alle persistenten Daten des Shops speichert werden.
2. gunicorn: Ein WSGI-Server, der die Django-Anwendung ausführt.
3. nginx: Webserver, welcher die statischen Dateien des Shops bereitstellt und als Reverse Proxy für Gunicorn fungiert.

Django besitzt zwar einen eingebauten Entwicklungsserver, jedoch ist dieser nicht für den Produktivbetrieb ausgelegt, da er nicht effizient mit mehreren Anfragen umgehen kann und keine optimale Performance bietet. Gunicorn ist ein WSGI-Server (Python Web Server Gateway Interface), der speziell für Python-Webanwendungen entwickelt wurde. Es bietet eine signifikant verbesserte Performance, da mehrere parallele Worker-Prozesse gestartet werden, welche mehrere Benutzeranfragen gleichzeitig verarbeiten können.

5.3. Hetzner-API

Wenn Benutzer einen Shop erstellen, muss dafür eine Domain angegeben werden. Hierbei steht zur Auswahl, ob diese intern oder extern verwaltet werden soll. Extern bedeutet, dass z.B. der Kunde die Domain konfiguriert und mit einem A-Record auf unsere Server zeigen lässt. Meistens wird jedoch eine interne Domain angegeben. Hetzner erlaubt die Konfiguration von Subdomains über eine API [2]. Dieses Python-Skript ermöglicht die automatisierte Konfiguration einer Subdomain:

```python
@hetzner_api
def configure_record(domain: str, zone: dict, name: str, type: str, value: str, ttl: int = 60) -> dict:
    target_record = {
            "value": value,
            "ttl": ttl,
            "type": type,
            "name": name,
            "zone_id": zone['id']
        }

    current_record = next(filter(lambda x: x['type'] == type and x['name'] == name,
                send_get_request('/records', {"zone_id": zone['id']})['records']), None)

    if current_record:
        # Record already configured
        if current_record['value'] != target_record['value']:
            target_record['id'] = current_record['id']
            return send_put_request(f'/records/{current_record['id']}', target_record)
    else:
        # Subdomain not yet configured
        return send_post_request('/records', target_record)
```

JSON-Daten welche an die API übergeben werden.

Überprüft ob die Subdomain bereits vorhanden ist.

Aktualisiert die Subdomain falls diese existiert.

Ansonsten wird sie neu erstellt.

5.4. Webshoperstellung durch Skripte

Zur Erstellung einer neuen Webshop-Instanz kommt Docker-Compose zum Einsatz. Mithilfe einer YAML-Konfigurationsdatei lassen sich dabei alle erforderlichen Ressourcen wie Anwendungscontainer, Datenbank, Caching-Dienste, Volumes und Netzwerke automatisiert bereitstellen und verwalten. Normalerweise verwendet man Docker-Compose indem man in einem Ordner eine Datei namens „docker-compose.yaml" erstellt. Alternativ kann man aber auch folgende Syntax verwenden:

```
docker compose -f - -p [project_name] up -d
```

Die Option „-f -“ ermöglicht es, die Docker-Compose-Konfiguration nicht aus einer Datei, sondern über die Standardeingabe (stdin) an den Befehl zu übergeben.

„-p [project_name]“ unterscheidet die verschiedenen Shops. Normalerweise wird der Projektname über den Verzeichnissnamen bestimmt. Hier wird dieser manuell angegeben.

Dieses Script erstellt einen neuen Shop mithilfe von Docker Compose:

```python
def up(project_name: str, compose: dict) -> int:
    process = subprocess.Popen(
        ["docker-compose", "-f", "-", "-p", project_name, "up", "-d"],
        stdin=subprocess.PIPE,
        text=True
    )
    compose_yaml = yaml.dump(compose, default_flow_style=False)
    process.communicate(input=compose_yaml)

    return process.returncode
```

Docker-Compose Prozess starten.

JSON-Konfiguration in YAML konvertieren.

Konfiguration über stdin an Docker-Compose übergeben.

Die Konfiguration wird als JSON anstatt in YAML generiert, da so Werte leichter überschrieben werden können. Diese Funktion erstellt ein Python Dictionary aus einem Template und einer Projektkonfiguration. Es werden mehrere Umgebungsvariablen an die Container übergeben, um diese zu konfigurieren:

```python
def generate_compose_dict(project: Project) -> dict:
    override_dict = {
        "services": {
            "gunicorn": {
                "environment": {
                    "DJANGO_SUPERUSER_PASSWORD":
project.config.django_superuser_password,
                    "DJANGO_SECRET_KEY": project.config.django_secret_key,
                    "DJANGO_ALLOWED_HOSTS": ','.join([project.config.domain, "localhost"]),
                    "DJANGO_TRUSTED_ORIGINS": f"https://{project.config.domain}",
                    "DJANGO_LANGUAGES": ','.join(project.config.languages),
                    "DATABASE_PASSWORD": project.config.db_password,
                },
            },
            "db": {
                "environment": {
                    "POSTGRES_PASSWORD": project.config.db_password,
                },
            },
            "nginx": {
                "labels": {
                    "traefik.enable": True,
                    f"traefik.http.routers.{project.name}-auto.rule": f"Host(`{project.config.domain}`)",
                },
            },
        },
    }
    if project.config.image:
        override_dict["services"]["gunicorn"]["image"] = project.config.image

    with open('/templates/template.json', 'r') as f:
        template_dict = json.load(f)
```

```
32
33      compose_dict = merge({}, template_dict, override_dict)
34
35      return compose_dict
```

In Zeile 30 wird mithilfe der Bibliothek „mergedeep“ das Python-Dictionary mit dem JSON-Template (Anhang 9.4) zusammengeführt.

Zusätzlich wurden Funktionen implementiert, mit denen sich Shops neu starten, löschen, den Status abfragen, Logs anzeigen oder einen Shop inklusive aller Volumes klonen lassen.

5.5. Datenstruktur

MongoDB wurde als Datenbank gewählt, da durch die Schemafreiheit heterogene Datensätze flexibel gespeichert werden können. Dies ist insbesondere deshalb von Vorteil, weil nicht alle Shops dieselbe Konfigurationsstruktur aufweisen müssen. Die dokumentenbasierte Struktur von MongoDB ermöglicht es, individuelle Einstellungen je Shop ohne starre Vorgaben abzubilden. Alle Shops werden in der Collection „projects“ gespeichert. Der Name kommt von Docker-Compose, da eine Shop-Instanz ein Compose Projekt ist. Das eingebettete Dokument „config“ speichert mehrere Konfigurationsdaten, die als Umgebungsvariablen an den Shop übergeben werden.

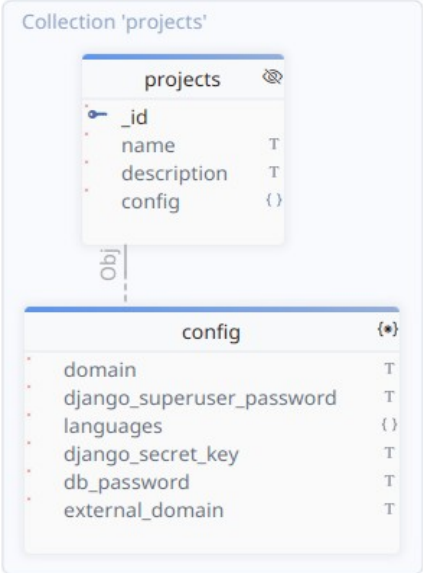

Abbildung 5: Datenstruktur

5.6. API

Für das Backend des Shop-Managers wird FastAPI verwendet.

Die vorherigen Scripte für DNS und Compose wurden in Python-Module umgewandelt. Diese werden von main.py verwendet, um Aktionen auszuführen. Sollten sich diese Schnittstellen in der Zukunft ändern, können sie einfach ausgetauscht werden. Dadurch wird die Wartbarkeit sichergestellt.

Hier wird beispielsweise die Funktion up() aus 5.4 verwendet, wenn der Endpunkt „/projects/{project_id}/up“ aufgerufen wird:

```
1 @app.get("/projects/{project_id}/up", response_model=dict)
2 async def run_project_up(project_id: str) -> dict:
3     project = await db.projects.find_one({"_id": ObjectId(project_id)})
4     if not project:
5         raise HTTPException(status_code=404, detail="Project not found")
6
7     compose_dict = compose.generate_compose_dict(Project(id=str(project["_id"]), **project))
8     compose.up(project["name"], compose_dict)
9     return {"message": "Befehl erfolgreich ausgeführt"}
```

Auch die API sollte als Container laufen. Dafür wurde ein Image erstellt und mit Docker Compose gestartet. So funktionierte aber die Kommunikation mit Docker-Compose nicht mehr um die Shops zu verwalten, da Docker nur auf dem Host und nicht in dem Container installiert ist. Um mit dem Docker-Host zu kommunizieren, muss das Docker-Socket in den Container gemountet werden:

```
        volumes:
        - /var/run/docker.sock:/var/run/docker.sock
```

Mit dem Socket hat man Zugriff auf Docker, aber nicht auf Docker-Compose. Dafür muss die docker-cli in dem Container installiert werden. Dies wird mithilfe der folgenden Dockerfile erledigt:

```
1  FROM docker:28-cli AS docker-base
2
3  FROM python:3.12-slim
4
5  COPY --from=docker-base /usr/local/bin/docker /usr/local/bin/docker
6  COPY --from=docker-base /usr/local/bin/docker-compose /usr/local/bin/docker-compose
7
8  WORKDIR /app
9
10  COPY ./requirements.txt .
11
12  RUN pip install --no-cache-dir -r requirements.txt
13
14  COPY ./app .
15
16  EXPOSE 8000
17
18  CMD ["fastapi", "run", "main.py", "--workers", "3", "--port", "8000"]
```

In den Zeilen 5 und 6 werden die Binärdateien für Docker und Docker-Compose aus dem offiziellen Docker-Image in das Python-Image kopiert. Dadurch konnten über Docker-Compose Container auf dem Host erstellt werden. Es gab jedoch eine kleine Abweichung: Seit einiger Zeit wird die neue Syntax „docker compose“ ohne Bindestrich verwendet. Auf dem Docker-Host wurden daher Befehle ohne Bindestrich ausgeführt, während im Container weiterhin „docker-compose“ mit Bindestrich erforderlich ist, obwohl die neuste Version des Images verwendet wurde.

5.7. Frontend

Das Frontend wurde mit dem Flask-Framework entwickelt und nutzt die API, um Informationen anzuzeigen oder Aktionen auszuführen. Dieser Ausschnitt zeigt eine Route welche Informationen zu einem Shop liefert:

```
1  @main.route('/show/<project_id>')
2  @login_required
3  def show_project(project_id: str) -> str:
4      response = requests.get(f"{FASTAPI_BASE_URL}/projects/{project_id}", )
5      project = response.json() if response.status_code == 200 else {}
6      project['status'], project['status_class'], project['containers'] =
determine_shop_status(project['id'])
7      return render_template('shop.html', project=project)
```

Nachdem die Daten von der API abgerufen wurden, wird das Template „shop.html“ (Anhang 9.5) gerendert. Nachfolgend ein vereinfachter Ausschnitt:

```
<a href="{{ url_for('main.index') }}">Zurück zur Übersicht</a>
<h2>{{ project.name }}</h2>
<div>
  <a href="{{ url_for('main.show_logs', project_id=project.id) }}">Logs</a>
  <a href="{{ url_for('main.edit_project', project_id=project.id) }}">Bearbeiten</a>
```

Mithilfe von „{{ }}“ kann auf Variablen zugegriffen werden, welche der „render_template“ Funktion übergeben wurden. Die Funktion „url_for“ liefert die URL für eine Route. Dadurch müssen URLs nicht mehrfach im Code hinterlegt werden, sondern können zentral definiert und bei Bedarf schnell angepasst werden. „{{ url_for('show_project') }}" liefert z.B. die URL für die oben gezeigte Route ("/show/<project_id>").

5.7.1. Weboberfläche

Die Weboberfläche wurde mit Bootstrap entwickelt, wodurch eine schnelle Umsetzung mit einem ansprechendem Design ermöglicht wurde. In der folgenden Abbildung ist eine Shop-Instanz im Shop-Manager zu sehen:

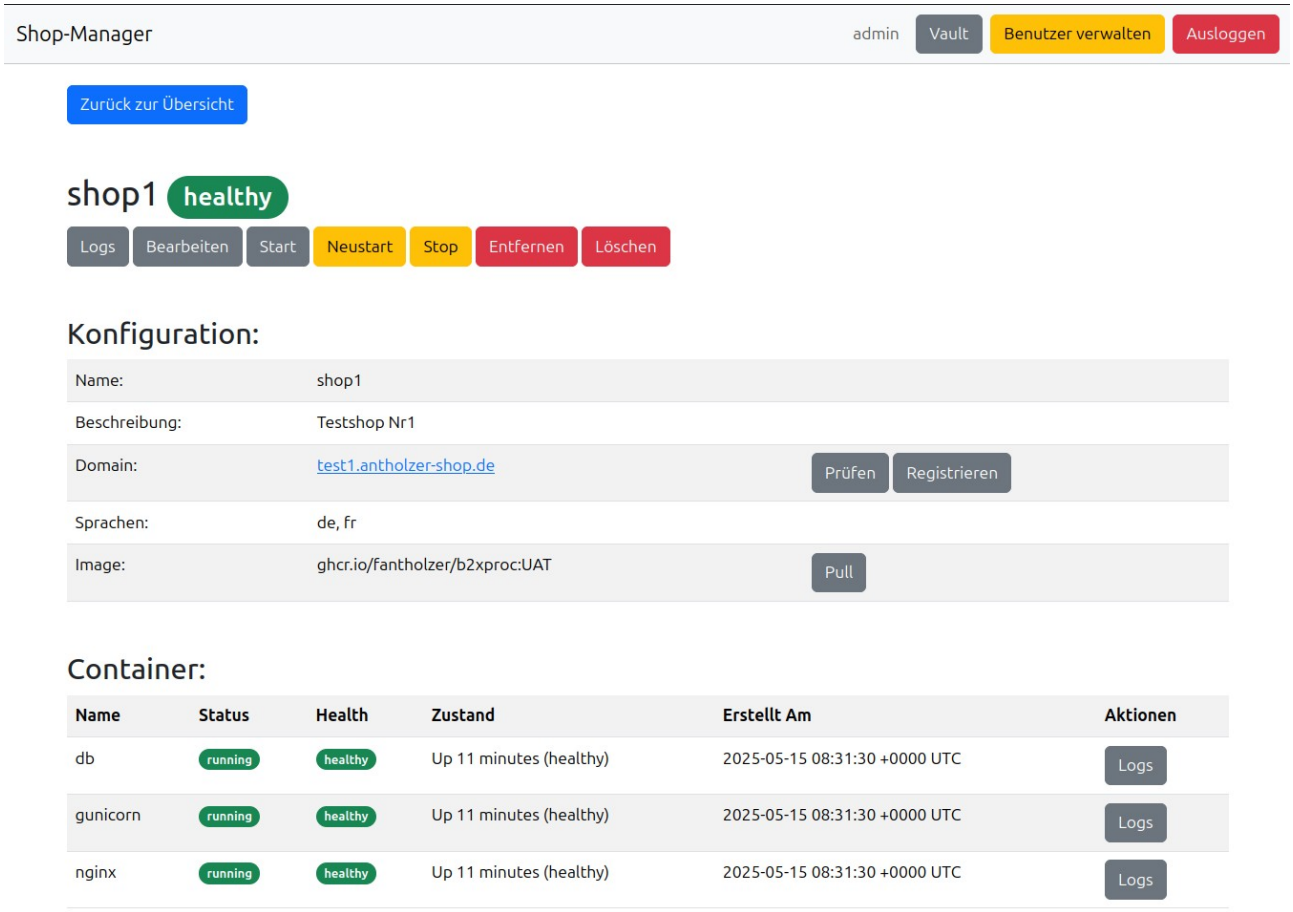

Abbildung 6: Ansicht eines Shops im Shop-Manager

Der Status der gesamten Shop-Instanz und der einzelnen Container kann in Form einer Ampel abgelesen werden. Man kann die Logs der Container ansehen, den Shop stoppen, starten und löschen. Außerdem lässt sich prüfen, ob die Domain korrekt konfiguriert ist. Falls diese intern verwaltet wird, kann sie bei Hetzner registriert werden.

Shops können auch wie in Abbildung 8 (Anhang 9.6) erstellt und bearbeitet werden. Durch die Checkbox „Nach dem Erstellen starten“ kann der Shop sofort nach dem Erstellen automatisch gestartet werden. Dadurch werden alle nötigen Schritte sofort ausgeführt, damit der Shop in wenigen Sekunden startet.

Abbildung 9 zeigt eine Übersicht aller Shops. Auch hier wurde das Ampelsystem verwendet, um mit einem schnellen Blick den Status aller Shops prüfen zu können.

5.7.2. Benutzerverwaltung

In Abbildung 10 ist die Benutzerverwaltung zu sehen. Diese ist sehr simpel gehalten, da man nur Benutzer erstellen und löschen kann. Beim Erstellen müssen Benutzername und Passwort sowie optional Administratorrechte gesetzt werden.

Für die Authentifizierung wurde die Bibliothek „Flask-Login“ verwendet. Da es nur zwei Benutzergruppen gibt, wird ausschließlich zwischen „Admin“ und „nicht Admin“ mit einem Boolean unterschieden:

```
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(150), unique=True, nullable=False)
    password = db.Column(db.String(150), nullable=False)
    is_admin = db.Column(db.Boolean, default=False)
```

Damit eine Route nur für authentifizierte Benutzer aufrufbar ist, kann der „@login _required“ Dekorator verwendet werden:

```
@main.route('/run/<project_id>/<action>')
@login_required
def run_action(project_id: str, action: str) -> Response:
    [...]
```

Eine Route, die ausschließlich für Administratoren zugänglich sein soll, kann mit „@admin_required“ abgesichert werden:

```
@main.route('/delete/shop/<project_id>')
@admin_required
def delete_project(project_id: str) -> Response:
    [...]
```

Der Dekorator „login_required“ wird von „Flask-Login“ bereitgestellt, „admin_required“ musste aber selbst entwickelt werden:

```python
def admin_required(f: callable):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if not current_user.is_authenticated:
            flash("You need to log in to access this page.", "warning")
            return redirect(url_for("auth.login"))

        if not current_user.is_admin:
            flash("You do not have permission to access this page.", "danger")
             return redirect(redirect_url())

         return f(*args, **kwargs)

     return decorated_function
```

Prüft ob der Nutzer authentifiziert ist.

Prüft ob der Nutzer autorisiert ist.

Bei Erfolg wird die Anfrage weiter bearbeitet.

5.8. Testphase

Während der Entwicklung wurden die Bereiche einzeln getestet. Nach der Fertigstellung des Shop-Managers wurden alle Funktionen zusammen getestet. Dank guter Fehlerbehandlung im Back- und Frontend wurden Fehler schnell identifiziert und anschließend sofort behoben.

```python
@app.get("/projects/{project_id}/down", response_model=dict)
async def run_project_down(project_id: str, delete: bool = False) -> dict:
    project = await db.projects.find_one({"_id": ObjectId(project_id)})
    if not project:
        raise HTTPException(status_code=404, detail="Shop nicht gefunden")
    compose.down(project["name"], delete)
    return {"message": "Befehl erfolgreich ausgeführt"}
```

In diesem Beispiel wird überprüft, ob der Shop existiert bevor Docker-Compose Befehle ausgeführt werden.

5.9. Monitoring

Traefik bietet viele Metrics die man mit Prometheus auslesen und anschließend mit Grafana auswerten kann. Zunächst muss die Konfiguration von Traefik angepasst werden (Anhang 9.2). Die genaue Bedeutung der Konfiguration kann in der Traefik Dokumentation nachgelesen werden [3].

```
[...]
16 metrics:
17   prometheus:
18     buckets:
19       - 0.1
20       - 0.3
21       - 1.2
22       - 5.0
23     addEntryPointsLabels: true
24     addServicesLabels: true
25     entryPoint: metrics
[...]
39 entryPoints:
[...]
53   metrics:
54     address: ":8899"
[...]
```

Aktiviert Metrics.

Port von dem Prometheus die Daten auslesen kann.

Danach wurde Prometheus so konfiguriert, dass es die Metrics von Traefik regelmäßig abruft:

```
1  global:
2    scrape_interval:     15s
3
4  scrape_configs:
5    - job_name: 'prometheus'
6
7      scrape_interval: 5s
8
9      static_configs:
10       - targets: ['traefik:8899']
```

Anschließend wurde in Grafana die Prometheus Instanz als Datenquelle angegeben und eine Dashboard-Vorlage für Traefik aus dem Internet in Grafana importiert. Nach ein paar Minuten füllten sich die Grafen:

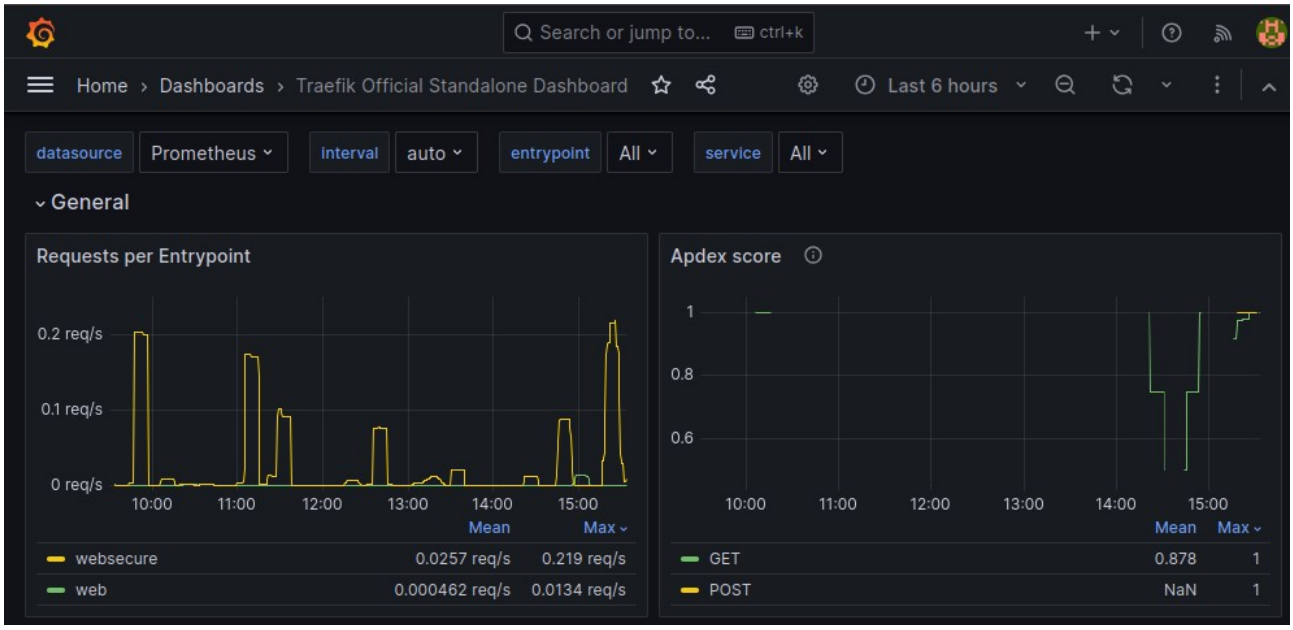

Abbildung 7: Grafana Dashboard

Zu sehen sind unter anderem die Anzahl der Anfragen und ein Apdex Score. Dieser wird automatisch von Grafana mithilfe der Daten aus Traefik berechnet. Der Apdex Score fasst das Nutzererlebnis in Bezug auf Ladezeiten in einer einzigen Zahl zusammen.

6. Abnahmephase

Nach der erfolgreichen Präsentation des fertigen Shop-Managers vor dem Projektbetreuer wurde dieser anschließend im Backoffice vorgestellt, da eine regelmäßige Nutzung durch das Team vorgesehen ist. Der Shop-Manager läuft derzeit nur auf dem UAT-Server, da das neue Shopsystem „b2Xproc“ noch in der Entwicklungsphase steckt.

7. Dokumentation

Jede Tätigkeit wurde in Joplin im Markdown-Format dokumentiert. Dabei wurden verwendete Befehle sowie relevante Quellen festgehalten. Die gesammelten Notizen flossen detailgetreu in dieses Dokument mit ein. Ein entsprechender Ausschnitt ist in Anhang 9.7 dargestellt.

Zudem wurde eine Anleitung verfasst, die eine selbstständige Bedienung des Shops-Managers durch das Backoffice ermöglicht. Ein Auszug daraus ist in Anhang 9.8 zu finden.

8. Fazit

8.1. Soll-/Ist-Vergleich

Das Projektziel wurde vollständig erreicht. Alle geplanten Funktionen konnten erfolgreich umgesetzt und auf dem UAT-Server in Betrieb genommen werden. Die angestrebte Automatisierung der Shop-Erstellung sowie die einfache Bedienbarkeit durch das Backoffice wurden wie vorgesehen realisiert. Die Lösung erfüllt die definierten Anforderungen und bietet zusätzlich eine flexible Basis für zukünftige Erweiterungen.

Der Auftraggeber zeigte sich mit dem Projektergebnis sehr zufrieden. Insgesamt entspricht der Ist-Zustand dem gewünschten Soll-Zustand und übertrifft diesen in einigen Punkten sogar.

Projektphase	Geplante Zeit	Benötigte Zeit
Einarbeitung	3 Stunden	3 Stunden
IST-Analyse	2 Stunden	2 Stunden
SOLL-Analyse	3 Stunden	3 Stunden
Einrichtung Docker mit Traefik	5 Stunden	3 Stunden
Dockerfile und Image erstellen	3 Stunden	3 Stunden
Skripte für Hetzner API	2 Stunden	2 Stunden
Webshoperstellung per Skripte	5 Stunden	5 Stunden
Bedienung per Weboberfläche	6 Stunden	6 Stunden
Testphase	6 Stunden	4 Stunden
Erstellung Dokumentation	5 Stunden	5 Stunden
Monitoring		4 Stunden
Gesamt	**40 Stunden**	**40 Stunden**

Die Einrichtung von Docker (5.1) und die Testphase (5.8) dauerten kürzer als geplant. Diese gesparte Zeit wurde in das Monitoring durch Prometheus und Grafana investiert.

8.2. Ausblick

Die Entwicklung von b2Xproc läuft zurzeit auf Hochtouren und soll bis Ende dieses Jahres abgeschlossen sein. Nach der Fertigstellung wird dieses Projekt verwendet, um die Shops für Kunden zu erstellen.

Das Grafana-Dashboard könnte auf einem Monitor in der IT-Abteilung dauerhaft angezeigt werden, um den Zustand der Shops immer im Blick zu haben. Mit Grafana wurde zudem nach Abschluss des Projektes ein Alerting-System eingerichtet. Sobald ein Shop nicht mehr verfügbar ist oder zu lange Antwortzeiten aufweist, wird die IT-Abteilung per Push-Nachricht informiert.

Die Access-Logs von Traefik könnten mit Tools wie GoAccess analysiert werden, um detaillierte Einblicke in den Traffic zu erhalten.

8.3. Persönliches Fazit

Dieses Projekt war sehr stark von Docker geprägt. Ich hatte bereits einige Erfahrung mit Docker gesammelt, jedoch verlieh mir dieses Projekt einen deutlich tieferen Einblick in containerisierte Umgebungen.

Flask hat sich für die Entwicklung des Frontends als die bessere Wahl erwiesen, da es leichtgewichtig und flexibel ist. Für größere Projekte wäre jedoch Django aufgrund seiner umfangreichen Funktionen und besseren Skalierbarkeit wahrscheinlich die geeignetere Wahl gewesen.

An diesem Projekt zu arbeiten hat mir sehr viel Spaß gemacht. Das Projektziel wurde erfolgreich erreicht und das Ergebnis ist mir sehr gut gelungen. Insgesamt war es eine äußerst bereichernde Erfahrung, bei der ich sowohl meine technischen Fähigkeiten als auch mein Verständnis für moderne Softwarearchitekturen deutlich erweitern konnte.

Abbildungsverzeichnis

Literaturverzeichnis

1: Docker Inc., Install Docker Engine on Ubuntu, Abgerufen am: 10.04.2025, https://docs.docker.com/engine/install/ubuntu/
2: Hetzner Online GmbH, Hetzner DNS Public API, Abgerufen am: 10.04.2025, https://dns.hetzner.com/api-docs
3: Traefik Labs, Prometheus, Abgerufen am: 02.05.2025, https://doc.traefik.io/traefik/observability/metrics/prometheus/

9. Anhang

9.1. Glossar

Apdex Score	Messwert für die Benutzerzufriedenheit, der die Performance einer Anwendung anhand der Antwortzeiten bewertet. Er basiert auf drei Kategorien: zufriedenstellend, tolerierbar und unbefriedigend. Der Apdex Score hilft dabei, die Nutzererfahrung objektiv zu analysieren. Ein Wert von 0 (Schlecht) bis 1 (Gut).
API	Schnittstelle über die verschiedene Software-Komponenten miteinander kommunizieren können. In Webprojekten werden häufig REST- oder GraphQL-APIs verwendet, um Daten zwischen Frontend und Backend auszutauschen.
Container	Isolierter, leichtgewichtiger Prozess, der eine Anwendung und ihre Abhängigkeiten enthält. Container laufen unabhängig vom Host-Betriebssystem und ermöglichen eine konsistente Umgebung unabhängig vom Host-Rechner.
Django	Python-Webframework, das sich durch seine hohe Produktivität und ein integriertes Admin-Interface auszeichnet. Es folgt dem Model-View-Template (MVT)-Architekturprinzip und bietet eine Vielzahl von Funktionen für die Webentwicklung, darunter ein ORM, Authentifizierung und Caching.

Docker	Plattform zur Containerisierung von Anwendungen. Mit Docker können Anwendungen samt Abhängigkeiten in Containern verpackt und auf beliebigen Systemen ausgeführt werden. Dies erleichtert die Bereitstellung und Skalierung.
Docker-Compose	Ein Werkzeug zur Definition und Verwaltung mehrerer Docker-Container als Anwendung. In einer YAML-Datei können verschiedene Container, Netzwerke, Volumes und Abhängigkeiten konfiguriert werden. Anschließend können alle Container mit einem einzigen Befehl gestartet, gestoppt oder skaliert werden.
Dockerfile	Eine Textdatei, die alle Anweisungen enthält, um ein Docker-Image zu erstellen. Darin werden Basis-Images, benötigte Software, Konfigurationsschritte und Befehle definiert, mit denen eine ausführbare Container-Umgebung aufgebaut wird.
Flask	Leichtgewichtiges Python-Webframework, das minimalistisch und flexibel ist. Im Gegensatz zu Django bietet Flask nur die Grundfunktionen für Webanwendungen und lässt sich durch Erweiterungen modular anpassen.
Framework	Ein vorgefertigtes Gerüst, das die Entwicklung von Software erleichtert und beschleunigt.
Object-Relational Mapping (ORM)	Technik zur Datenbankabstraktion, bei der Datenbanken über objektorientierte Klassen angesprochen werden, anstatt direkt mit SQL-Queries zu arbeiten. Django verwendet beispielsweise das Django ORM, während Flask häufig mit SQLAlchemy kombiniert wird.
PIP	Paketverwaltungsprogramm für Python-Pakete aus dem Python Package Index (PyPI).
Python-Modul	Wiederverwendbare Code-Datei in Python, die Funktionen, Klassen oder Variablen enthält. Python-Module ermöglichen eine modulare Entwicklung und können in anderen Skripten oder Projekten importiert werden.
SSL-Zertifikat	Ermöglicht eine verschlüsselte Verbindung zwischen Client und (Web-)Server. Zertifikate werden von Zertifizierungsstellen (CAs) wie Let's Encrypt ausgestellt und erhöhen die Sicherheit von Webanwendungen.
Traefik	Reverse Proxy und Load Balancer der speziell für containerisierte Umgebungen entwickelt wurde. Traefik kann automatisch Routing-Regeln aus Docker-Labels ableiten und integriert sich gut mit Let's Encrypt für automatische SSL-Zertifikate.

9.2. traefik.yml

```yaml
api:
  dashboard: true

log:
  level: INFO
  # filepath: /var/log/traefik.log

metrics:
  prometheus:
    buckets:
      - 0.1
      - 0.3
```

```
      - 1.2
      - 5.0
    addEntryPointsLabels: true
    addServicesLabels: true
    entryPoint: metrics

accessLog:
  filepath: /var/log/traefik-access.log

providers:
  docker:
    exposedByDefault: false
    network: proxy
  file:
    directory: /etc/traefik/dynamic
    watch: true


entryPoints:
  web:
    address: ":80"
    http:
      redirections:
        entryPoint:
          to: websecure
          scheme: https
  websecure:
    address: ":443"
    asDefault: true
    http:
      tls:
        certResolver: letsencrypt
  metrics:
    address: ":8899"

certificatesResolvers:
  letsencrypt:
    acme:
      email: heinrich.thaler@antholzer.de
      storage: "/letsencrypt/acme.json"
      tlsChallenge: {}
```

9.3. docker-compose.yaml

```
services:
  db:
    environment:
      POSTGRES_USER: odoo
      POSTGRES_DB: b2xproc_dev
      POSTGRES_PASSWORD: dbpassword9248
    image: postgres:17
    volumes:
    - postgres_data:/var/lib/postgresql/data
```

```
    restart: unless-stopped
    healthcheck:
      test: pg_isready -U odoo -d b2xproc_dev
      interval: 30s
      timeout: 5s
      retries: 5
  gunicorn:
    environment:
      DEBUG: false
      DJANGO_SETTINGS_MODULE: b2xproc.settings.production
      DJANGO_LOGLEVEL: info
      DATABASE_ENGINE: postgresql
      DATABASE_NAME: b2xproc_dev
      DATABASE_PORT: 5432
      DATABASE_USERNAME: odoo
      DATABASE_HOST: db
      DJANGO_SUPERUSER_PASSWORD: admin
      DJANGO_SECRET_KEY: "django-insecure-gG.&vnN_m<&%K_B+"
      DJANGO_ALLOWED_HOSTS: "*"
      DJANGO_TRUSTED_ORIGINS: "http://localhost"
      DJANGO_LANGUAGES: "de,en,fr,es"
      DATABASE_PASSWORD: dbpassword9248
    image: b2xproc
    depends_on:
    - db
    volumes:
    - static_volume:/static
    - media_volume:/media
    - gunicorn_data:/data
    restart: unless-stopped
    healthcheck:
      test: curl -f http://localhost:8000/health/
      interval: 30s
      timeout: 5s
      retries: 3
  nginx:
    image: nginx:1.27
    depends_on:
    - gunicorn
    volumes:
    - ./nginx-conf.d:/etc/nginx/conf.d:ro
    - static_volume:/var/www/static
    - media_volume:/var/www/media
    restart: unless-stopped
    healthcheck:
      test: curl -f http://localhost
      interval: 30s
      timeout: 5s
      retries: 3
    ports:
      - 80:80
volumes:
  postgres_data:
```

```
  static_volume:
  media_volume:
  gunicorn_data:
```

9.4. template.json

```
{
 "services": {
  "db": {
   "environment": {
    "POSTGRES_USER": "odoo",
    "POSTGRES_DB": "b2xproc_dev"
   },
   "image": "postgres:17",
   "volumes": [
    "postgres_data:/var/lib/postgresql/data"
   ],
   "networks": [
    "internal"
   ],
   "restart": "unless-stopped",
   "healthcheck": {
    "test": "pg_isready -U odoo -d b2xproc_dev",
    "interval": "30s",
    "timeout": "5s",
    "retries": 5,
    "start_period": "60s",
    "start_interval": "5s"
   }
  },
  "gunicorn": {
   "environment": {
    "DEBUG": "False",
    "DJANGO_SETTINGS_MODULE": "b2xproc.settings.production",
    "DJANGO_LOGLEVEL": "info",
    "DATABASE_ENGINE": "postgresql",
    "DATABASE_NAME": "b2xproc_dev",
    "DATABASE_PORT": "5432",
    "DATABASE_USERNAME": "odoo",
    "DATABASE_HOST": "db"
   },
   "image": "b2xproc",
   "deploy": {
    "replicas": 5
   },
   "depends_on": [
    "db"
   ],
   "volumes": [
    "static_volume:/static",
    "media_volume:/media",
    "gunicorn_data:/data"
   ],
```

```
    "networks": [
      "internal"
    ],
    "restart": "unless-stopped",
    "healthcheck": {
      "test": "curl -f http://localhost:8000/health/",
      "interval": "30s",
      "timeout": "5s",
      "retries": 3,
      "start_period": "5m",
      "start_interval": "5s"
    }
  },
  "nginx": {
    "image": "nginx:1.27",
    "depends_on": [
      "gunicorn"
    ],
    "volumes": [
      "/root/shop-manager/nginx-conf.d:/etc/nginx/conf.d:ro",
      "static_volume:/var/www/static",
      "media_volume:/var/www/media"
    ],
    "networks": [
      "internal",
      "proxy"
    ],
    "restart": "unless-stopped",
    "healthcheck": {
      "test": "curl -f http://localhost",
      "interval": "30s",
      "timeout": "5s",
      "retries": 3,
      "start_period": "60s",
      "start_interval": "5s"
    }
  }
},
"volumes": {
  "postgres_data": null,
  "static_volume": null,
  "media_volume": null,
  "gunicorn_data": null
},
"networks": {
  "internal": {
    "driver": "bridge",
    "internal": true
  },
  "proxy": {
    "external": true
  }
}
```

```
}
```

9.5. shop.html

```
{% extends "base.html" %}

{% block content %}
<a class="btn btn-primary mt-1 mb-5" href="{{ url_for('main.index') }}">Zurück zur Übersicht</a>
<h2>{{ project.name }} <span class="badge rounded-pill text-bg-{{ project.status_class }}">{{ project.status }}</span></h2>
<div class="actions my-2">
    <a class="btn btn-secondary" href="{{ url_for('main.show_logs', project_id=project.id) }}">Logs</a>
    <a class="btn btn-secondary" href="{{ url_for('main.edit_project', project_id=project.id) }}">Bearbeiten</a>
    <a class="btn btn-secondary" href="{{ url_for('main.run_action', project_id=project.id, action='up') }}">Start</a>
    <a class="btn btn-warning" href="{{ url_for('main.run_action', project_id=project.id, action='restart') }}">Neustart</a>
    {% if current_user.is_admin %}
        <a class="btn btn-warning" href="{{ url_for('main.run_action', project_id=project.id, action='down') }}">Stop</a>
        <a class="btn btn-danger" href="{{ url_for('main.remove_project', project_id=project.id) }}">Entfernen</a>
        <a class="btn btn-danger" href="{{ url_for('main.delete_project', project_id=project.id) }}">Löschen</a>
    {% endif %}
</div>
<h3 class="mt-5">Konfiguration:</h3>
<table class="table table-striped">
    <tr>
        <td>Name:</td>
        <td>{{ project.name }}</td>
        <td></td>
    </tr>
    <tr>
        <td>Beschreibung:</td>
        <td>{{ project.description }}</td>
        <td></td>
    </tr>
    <tr>
        <td>Domain:</td>
        <td><a href="https://{{ project.config.domain }}" target="_blank" id="domain">{{ project.config.domain }}</a>{% if project.config.external_domain %} (Extern){% endif %}</td>
        <td>
            <a class="btn btn-secondary" onclick="checkDomain()">Prüfen</a>
            {% if not project.config.external_domain %}
            <a class="btn btn-secondary" href="{{ url_for('main.update_domain', domain=project.config.domain) }}">Registrieren</a>
            {% endif %}
        </td>
    </tr>
```

```
39      <tr>
40          <td>Sprachen:</td>
41          <td>{{ project.config.languages|join(", ") }}</td>
42          <td></td>
43      </tr>
44  </table>
45  <h3 class="mt-5">Container:</h3>
46  <table class="table table-striped">
47      <tr>
48        <th>Name</th>
49        <th>Status</th>
50        <th>Health</th>
51        <th>Zustand</th>
52        <th>Erstellt Am</th>
53        <th>Aktionen</th>
54      </tr>
55      {% for container in project.containers %}
56          <tr>
57              <td>{{ container.service }}</td>
58              <td><div class="badge rounded-pill text-bg-{{ container.state_class }}">{{
container.state }}</div></td>
59              <td><div class="badge rounded-pill text-bg-{{ container.health_class }}">{{
container.health }}</div></td>
60              <td>{{ container.status }}</td>
61              <td>{{ container.createdat }}</td>
62              <td><a class="btn btn-secondary" href="{{ url_for('main.show_logs',
project_id=project.id, service=container.service) }}">Logs</a></td>
63          </tr>
64      {% endfor %}
65  </table>
66  <script>
67      function checkDomain() {
68          let domain = document.getElementById("domain").innerText.trim();
69          if (!domain) {
70              alert("Bitte geben Sie eine Domain ein.");
71              return
72          }
73          fetch("{{ url_for('main.check_domain', domain='') }}".replace('//', '/' +
encodeURIComponent(domain) + '/'))
74          .then(response => response.json())
75          .then(data => {
76              if (data.success) {
77                  showMessage(data.message, "success");
78              } else {
79                  showMessage(data.message, "danger");
80              }
81          })
82          .catch(error => {
83              showMessage("Fehler beim Überprüfen der Domain!", "danger");
84          });
85      }
86  </script>
87  {% endblock %}
```

9.6. Screenshots vom Shop-Manager

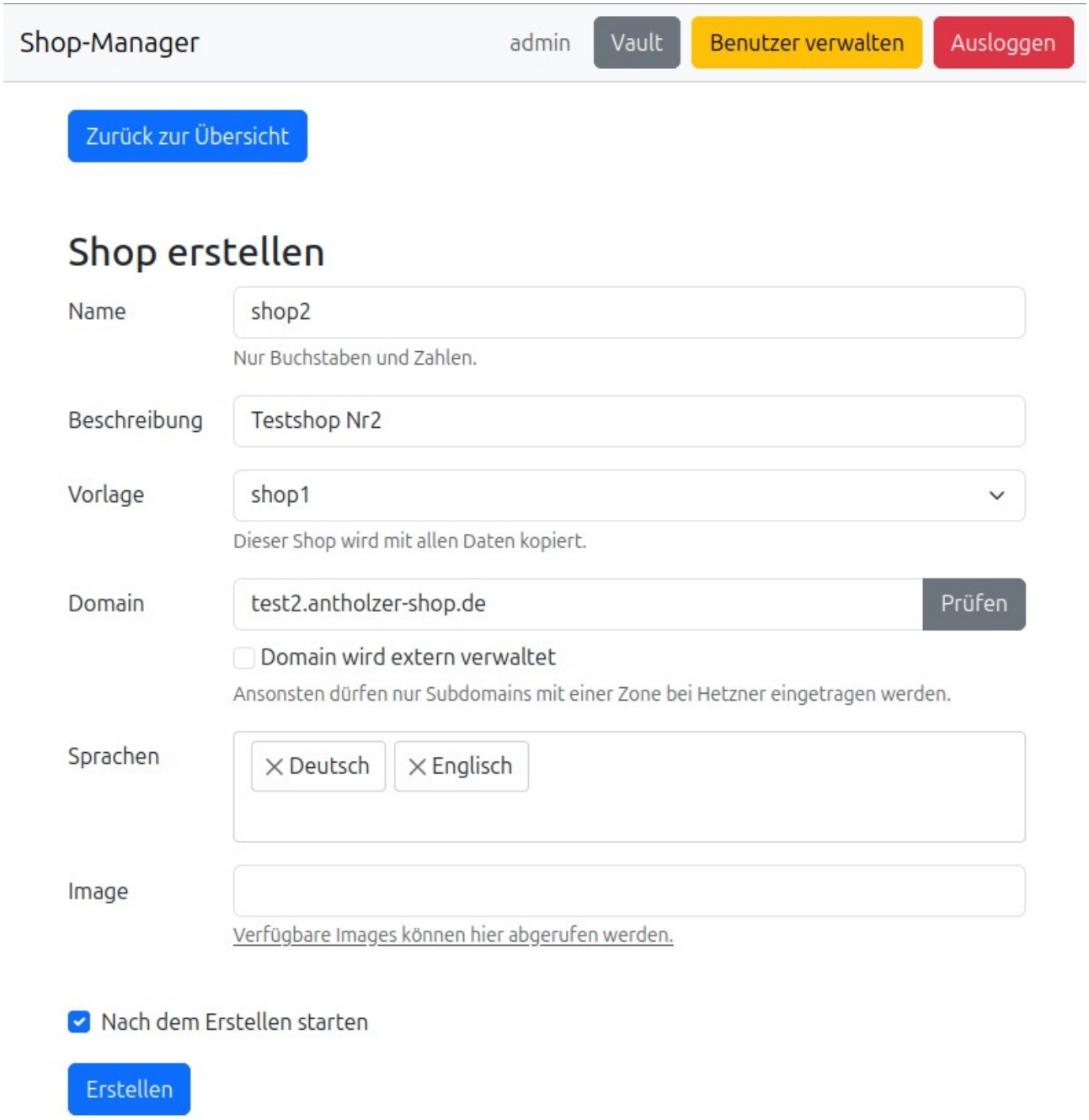

Abbildung 8: Formular zur Erstellung eines Shops

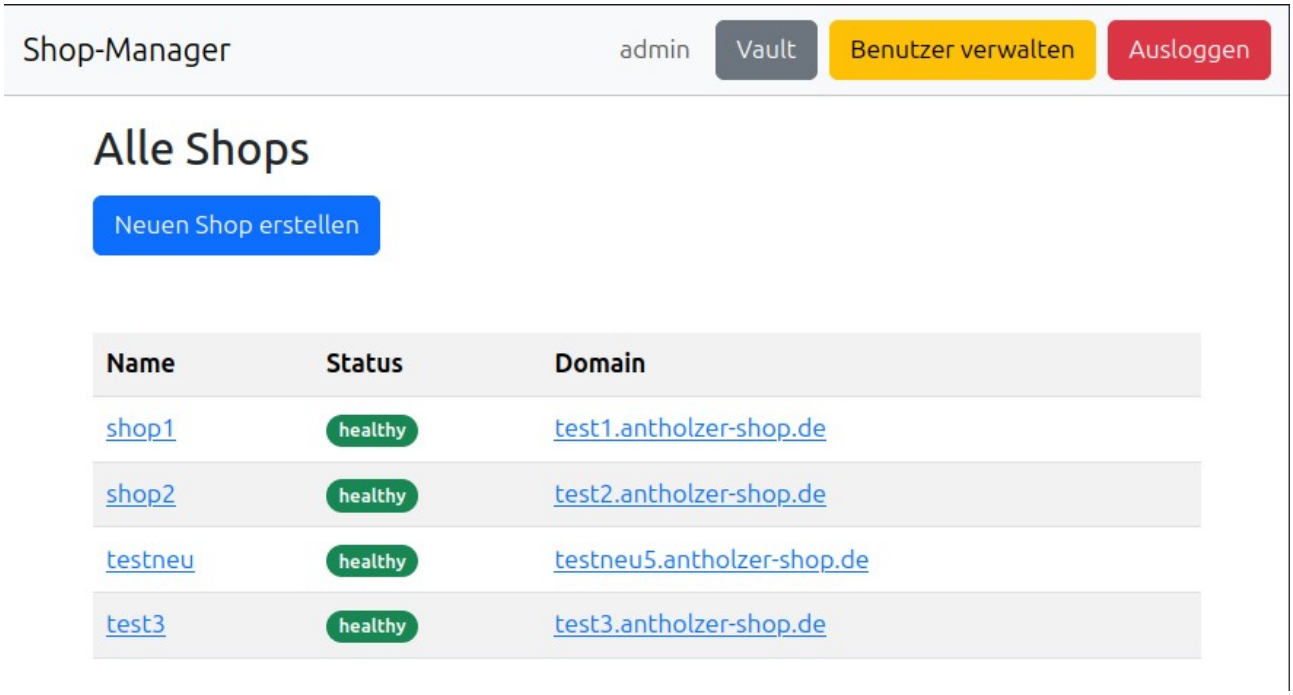

Abbildung 9: Übersicht über alle Shops

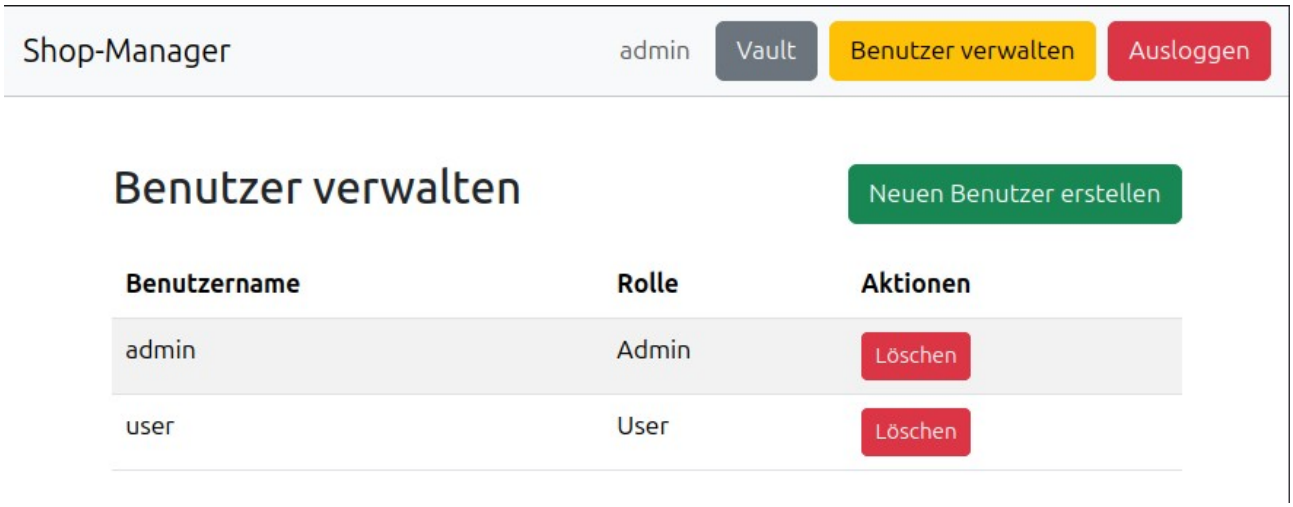

Abbildung 10: Benutzerverwaltung

9.7. Ausschnitt Mitschrift

Domain registrieren

Überprüfen:

`dig example.com NS`

`dig example.com +trace`

`dig @docks10.rzone.de. +short mcusw.de`

Traffic + Hetzner dnsChallange

https://doc.traefik.io/traefik/https/acme/#dnschallenge

API Aufruf

Hetzner DNS API

https://dns.hetzner.com/api-docs

Zone finden

`curl "https://dns.hetzner.com/api/v1/zones" -H 'Auth-API-Token: $token'`

Bestehende Records holen

`curl "https://dns.hetzner.com/api/v1/records?zone_id={ZoneID}" -H 'Auth-API-Token: $token'`

Record erstellen

```
curl -X "POST" "https://dns.hetzner.com/api/v1/records" \
     -H 'Content-Type: application/json' \
     -H 'Auth-API-Token: $token' \
     -d $'{
  "value": "1.1.1.1",
  "ttl": 86400,
  "type": "A",
  "name": "www",
  "zone_id": "1"
}'
```

Abbildung 11: Ausschnitt aus der Joplin Mitschrift

9.8. Ausschnitt Benutzerdokumentation

Feld	Beschreibung
Name	Name des Shops der in der Übersicht zu sehen ist. Nur Buchstaben und Zahlen erlaubt.
Beschreibung	Beliebige Beschreibung des Shops.
Vorlage	Ein Shop kann als Vorlage verwendet werden. Dabei werden ALLE Daten des Shops kopiert.
Domain	Die Domain unter der der Shop erreichbar sein soll.
Domain wird extern verwaltet	z.B. hat der Kunde seine eigene Domain, welche er verwenden möchte. Die Domain muss auf unsere Server zeigen. Das kann mit dem „Prüfen“ Knopf überprüft werden.
Sprachen	Diese Sprachen können im Shop ausgewählt werden.
Nach dem Erstellen starten	Nach dem Erstellen des Shops werden alle anderen nötigen Schritte damit der Shop erreichbar wird automatisch ausgeführt.

Nachdem ein Shop erstellt wurde, sind die Container im Status „starting“ (gelb). Innerhalb einer Minute sollten diese auf „healthy“ (grün) wechseln. Ab dann ist der Shop erreichbar.

Abbildung 12: Ausschnitt aus der Benutzerdokumentation

Heinrich Thaler
16.05.2025
www.it-thaler.de
info@it-thaler.de