



26.05.2024 - CHALLENGE TASK FRÜHLINGSSEMESTER 2024

Site Pulse 360

Distributed Systems

STUDENTEN: **Valentino Diller**
valentino.diller@ost.ch

Andrin Kessler
andrin.kessler@ost.ch

Daniel Schatzmann
daniel.schatzmann@ost.ch

DOZENT: **Dr. Thomas Bocek**
thomas.bocek@ost.ch

List of contents

1 Architektur	1
1.1 Techstack	1
1.2 Netzwerk	1
1.3 Begründung Cilium VIP / Traefik	2
2 Installation	3
2.1 Installation VMs	3
2.2 Installation K3s	4
2.3 Installation Cilium	5
2.4 Etd	7
2.5 PostgreSQL	8
2.6 Patroni	8
2.7 HAProxy	10
2.8 GitLab Agent	11
2.9 GitLab Runner	11
2.10 Gitlab CI/CD	12
3 Applikation	14
3.1 Frontend	14
3.2 Backend	16
3.3 Ausführen der Applikation	20
4 Fazit	21
A Abbildungsverzeichnis	22

1 Architektur

1.1 Techstack

Ziel ist es, eine hochverfügbare, dockerisierte Applikation zu erstellen. Dies wird mit einem Kubernetes-Cluster erzielt. Die einzelnen Nodes werden für dieses Projekt auf einem VMware-ESXi Hypervisor virtualisiert und laufen mit Ubuntu als Betriebssystem. Die hochverfügbare IP und das interne Netzwerk des Clusters wird durch das Cilium-Plugin bereitgestellt.

Um die Datenbank sowohl persistent als auch hochverfügbar zu machen, wird ein Patroni-Cluster verwendet. Dieser basiert auf PostgreSQL und der Cluster wird über die selben Nodes gespannt, die auch den Kubernetes-Cluster bilden. Die Anfragen an die Datenbank werden durch HAProxy an den Leader des Clusters weitergeleitet.

Die Applikation wird durch eine GitLab CI/CD Pipeline gebaut und deployed. Als Base-Image der Container wird Alpine Linux verwendet. Das Frontend wird mittels Vue.js umgesetzt, im Backend wird Golang eingesetzt. Traefik verwaltet die Authentifizierung und leitet die Anfragen basierend auf dem Pfad an den jeweiligen Pod weiter.

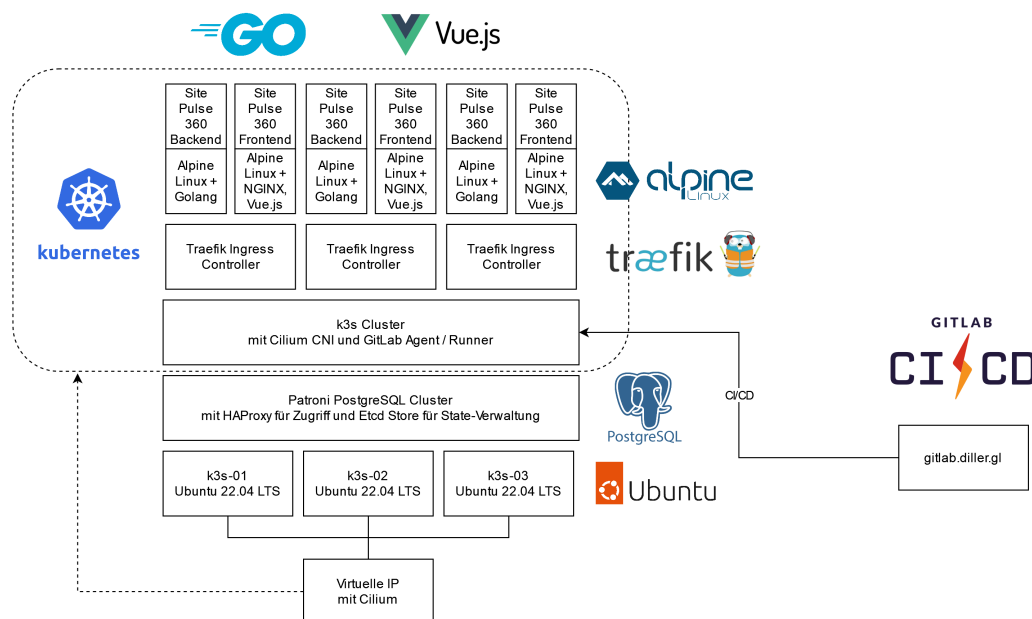


Figure 1: Visualisierung Techstack + Aufbau

1.2 Netzwerk

Für die Netzwerk-Infrastruktur wurde eine bereits vorhandene Umgebung verwendet. Diese besteht aus einer OPNsense-Firewall und einem vorinstallierten ESXi-Hypervisor. Um das Sitepulse-Netzwerk davon abzugrenzen, wurde ein neues VLAN (ID 1010) mit der Netzwerkadresse 10.0.10.0/24 erstellt. Die Firewall leitet die HTTP(S)-Anfragen mit der URL sitepulse.future-solutions.ch mittels eines NGINX Reverse-Proxy an die IP 10.0.10.10, welche durch einen der Nodes vertreten wird, weiter.

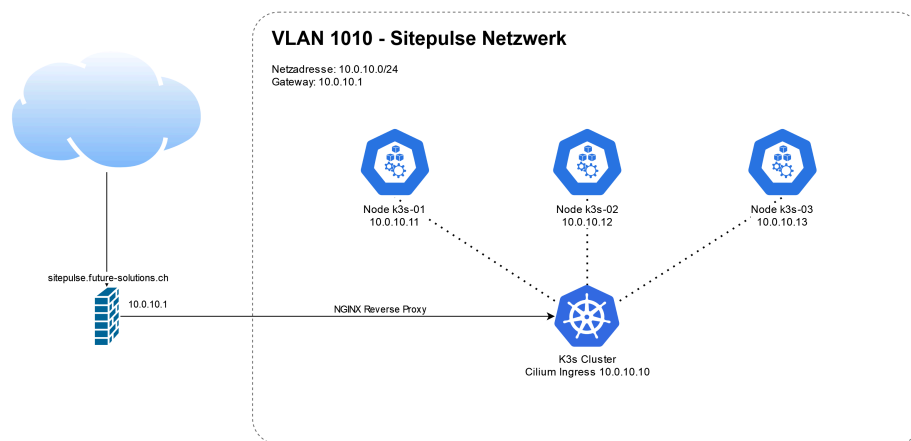


Figure 2: Netzwerkplan Sitepulse Netzwerk

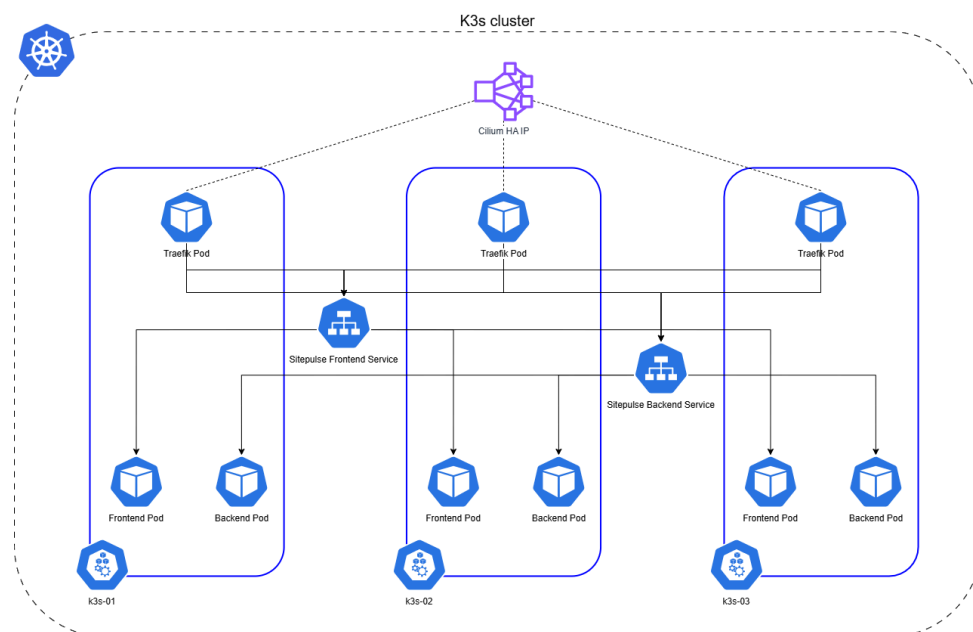


Figure 3: K3s Cluster Aufbau

1.3 Begründung Cilium VIP / Traefik

In unserer Architektur hätte man sich grundsätzlich die virtuelle IP-Adresse von Cilium und den Traefik Ingress Controller sparen können. Die Frontend und Backend Pods könnten direkt vom Nginx Reverse Proxy über einen NodePort bei den Nodes angesprochen werden. Nginx hätte alle Funktionalitäten des Traefik Ingress Controllers übernehmen können (Healthcheck, Basic Auth, Path- und Method-based Routing).

Jedoch wollten wir das Setup so realitätsnah wie möglich gestalten. In der Praxis hat man (sofern keine WAF im Einsatz ist) keinen Reverse Proxy zusätzlich vor den Ingress Controller geschaltet. Der Ingress Controller ist in der Regel direkt über eine virtuelle IP-Adresse erreichbar, so ist es nun auch beim Setup dieses Projekts. Wir konnten so auch etwas dazulernen bei der Konfiguration von Cilium und Traefik mit Kubernetes CRDs. Beim Zugriff aus dem Internet über <https://sitepulse.future-solutions.ch/> geht man einfach zuerst via Nginx Reverse Proxy, welcher Name-based Routing macht, auf die virtuelle IP-Adresse von Traefik.

2 Installation

2.1 Installation VMs

Der erste Schritt der Umsetzung unseres Challenge Tasks war, die drei VMs mit den Hostnames k3s-01, k3s-02 und k3s-03 aufzusetzen.

2.1.1 Distribution Ubuntu 22.04

Wir haben uns für die Distribution Ubuntu 22.04 entschieden, weil sie K3s, etcd, PostgreSQL und Patroni offiziell unterstützt und Pakete dafür in den Repositories hat. Ausserdem wird keine Lizenz benötigt und wir haben bereits Erfahrung damit. Versionsmässig haben wir die zu diesem Zeitpunkt neuste Long Term Support-Version gewählt, um die neusten Features und Sicherheitsupdates zu erhalten.

2.1.2 Hardware

Die VMs wurden mit 2 CPUs, 4GB RAM, einer Systemdisk mit 32GB und einer Datendisk mit 10GB ausgestattet. Somit sind alle Mindestanforderungen erfüllt und wir haben noch etwas Spielraum für die Applikation.

2.1.3 Installation

Während der Installation ab ISO haben wir mit dem Subiquity Installer die normale Server Installation ausgewählt, den OpenSSH-Server aktiviert und einen User namens k3sadm mit Sudo-Rechten erstellt. Ausserdem haben wir folgendes Custom Disk Layout erstellt:

DEVICE		TYPE	SIZE	
[vg_system (new)		LVM volume group	29.945G	▶]
lv_root	new, to be formatted as ext4, mounted at /		8.000G	▶
lv_var	new, to be formatted as ext4, mounted at /var		10.000G	▶
lv_var_log	new, to be formatted as ext4, mounted at /var/log		10.000G	▶
lv_tmp	new, to be formatted as ext4, mounted at /tmp		1.000G	▶
[vg_data (new)		LVM volume group	9.996G	▶]
lv_postgres	new, to be formatted as ext4, mounted at /opt/postgres		9.996G	▶
[/dev/sda		local disk	32.000G	▶]
partition 1	new, primary ESP, to be formatted as fat32, mounted at /boot/efi		1.049G	▶
partition 2	new, to be formatted as ext4, mounted at /boot		1.000G	▶
partition 3	new, PV of LVM volume group vg_system		29.948G	▶
[/dev/sdb (PV of LVM volume group vg_data)		local disk	10.000G	▶]

Figure 4: Ubuntu Disk Layout

Es gibt somit separate Partitionen bzw. Logical Volumes für folgende Verzeichnisse:

- /boot/efi: Wird automatisch erstellt und wird benötigt, damit das System von UEFI booten kann.
- /boot: Enthält die Kernel und das initiale RAM-Dateisystem.
- /: Enthält das Betriebssystem und die Programme.
- /var: Enthält variable Daten, wie K3s Container und Images oder Caches.
- /var/log: Enthält Logfiles.
- /tmp: Enthält temporäre Dateien.
- /opt/postgres: Enthält die Postgres Datenbank.

Dank LVM sind wir sehr flexibel und können einfach Änderungen am Layout vornehmen.

2.2 Installation K3s

Der zweite Schritt der Umsetzung unseres Challenge Tasks war, einen K3s Cluster mithilfe der drei Nodes zu bauen. Wir haben uns für K3s entschieden, weil es sehr leichtgewichtig ist und trotzdem alle unsere Anforderungen erfüllt. Ausserdem ist es weit verbreitet und die Installation ist sehr einfach.

2.2.1 Vorbereitungen

Zuerst haben wir Ubuntu auf den neusten Stand gebracht:

```
1 apt update && apt upgrade -y && apt autoremove -y && reboot
```

Dann haben wir die Uncomplicated Firewall aktiviert und konfiguriert:

```
1 ufw enable
2 ufw allow 6443/tcp                # Kubernetes API Server
3 ufw allow from 10.42.0.0/16 to any # Cilium Pod Netzwerk
4 ufw allow from 10.43.0.0/16 to any # Cilium Service Netzwerk
5 ufw allow 2379:2380/tcp           # Etcd
6 ufw allow 10250/tcp               # Kubelet
7 ufw allow 8472/udp                # Cilium VXLAN
8 ufw allow 4240/tcp                # Cilium Cluster Health Checks
9 ufw allow 4244/tcp                # Cilium Hubble Server
10 ufw allow 4245/tcp               # Cilium Hubble Relay
11 ufw allow 4250/tcp               # Cilium Mutual Authentication port
12 ufw allow 9962/tcp               # Cilium Agent Prometheus Metrics
13 ufw allow 9963/tcp               # Cilium Operator Prometheus Metrics
```

Als einheitlichen Hostname für die Kubernetes API haben wir k3s gewählt. Momentan löst er einfach lokal auf die IP vom ersten Node auf, später hätten wir aber die Möglichkeit, einen Load Balancer davor zu schalten, um die API hochverfügbar zu machen.

```
1 echo '10.0.10.11 k3s' >> /etc/hosts
```

2.2.2 Konfiguration

Zwecks Nachvollziehbarkeit und Wartbarkeit haben wir die ganze K3s Konfiguration im File `/etc/rancher/k3s/config.yaml` auf den drei VMs vorgenommen:

config.yml

```

1 # Neuen Cluster erstellen (nur auf k3s-01)
2 cluster-init: true
3 # Zu kontaktierende Kubernetes API für Cluster Join (nur auf k3s-02 und k3s-03)
4 server: https://k3s:6443
5 # Nur der User root kann auf die Kubeconfig zugreifen, Helm würde eine Fehlermeldung
  anzeigen, wenn die Kubeconfig group-readable oder world-readable ist.
6 write-kubeconfig-mode: "0600"
  # Der API Hostname, den wir in der /etc/hosts Datei eingetragen haben, soll im
7 API Zertifikat als Subject Alternative Name (SAN) eingetragen werden, damit die
  API auch über den Hostname ohne Zertifikatsfehler erreichbar ist.
8 tls-san:
9   - "k3s"
10 # Der Token, den wir auf k3s-01 generiert haben und k3s-02/k3s-03 verwenden für
  den Cluster Join.
11 token: 'supersecret'
12 # Flannel wird standardmässig installiert, wir verwenden aber die modernere
  Alternative Cilium, daher wird Flannel deaktiviert.
13 flannel-backend: none
14 # ServiceLB wird standardmässig installiert, jedoch bringt Cilium diese
  Funktionalität schon mit, daher wird ServiceLB deaktiviert.
15 disable:
16   - servicelb
17 # Network Policies und der Kube Proxy können deaktiviert werden, weil Cilium diese
  Funktionalitäten übernimmt.
18 disable-network-policy: true
19 disable-kube-proxy: true

```

2.2.3 Installation

Kubernetes benötigt eine ungerade Anzahl ≥ 3 an Master Nodes ("server" in K3s), um ein hochverfügbares Control plane zu gewährleisten. Wir haben uns für 3 Nodes entschieden, um die minimalste Anzahl an Nodes zu verwenden, die ein hochverfügbares Control plane ermöglicht. In der Produktion würde man keinen Workload auf den Master Nodes laufen lassen, sondern nur auf den Worker Nodes. Da wir aber die Anzahl VMs möglichst niedrig halten wollten, haben wir uns für 3 Nodes entschieden, auf denen sowohl Master als auch Worker Rollen laufen. Mit diesem Command wird K3s mit Master und Worker Rolle auf den Nodes installiert:

```
1 curl -sL https://get.k3s.io | INSTALL_K3S_EXEC="server" sh -
```

Die Konfiguration aus dem File `/etc/rancher/k3s/config.yml` wird automatisch miteinbezogen.

2.3 Installation Cilium

Der dritte Schritt der Umsetzung unseres Challenge Tasks war, das Cilium CNI Plugin zu installieren, um ein internes Netzwerk für die Pods im Cluster zu erstellen. Die Schritte mussten nur auf dem ersten Node durchgeführt werden, da Kubernetes Konfigurationen für den ganzen Cluster gelten. Wir haben uns für Cilium anstatt dem Standard Netzwerk-Plugin Flannel entschieden, weil es keinen Kube-Proxy benötigt und wir somit noch mehr Ressourcen sparen. Ausserdem hat es eine sehr gute Performance, Network Policies wenn gewünscht, eine grosse Verbreitung und bei Bedarf auch fortgeschrittene Observability Tools.

2.3.1 Vorbereitungen

Zuerst mussten wir den Kubernetes Paket Manager Helm installieren mithilfe des Ubuntu Paket Managers (Commands aus offizieller Helm Anleitung):

```

1 curl https://baltocdn.com/helm/signing.asc | gpg --dearmor | sudo tee /usr/share/
  keyrings/helm.gpg > /dev/null
2 sudo apt-get install apt-transport-https --yes
  echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/
3 helm.gpg] https://baltocdn.com/helm/stable/debian/ all main" | sudo tee /etc/apt/
  sources.list.d/helm-stable-debian.list
4 sudo apt-get update
5 sudo apt-get install helm

```

Dann haben wir Helm auf den K3s Cluster verwiesen und das Cilium Helm Repository hinzugefügt:

```

1 export KUBECONFIG=/etc/rancher/k3s/k3s.yaml
2 helm repo add cilium https://helm.cilium.io/
3 helm repo update

```

2.3.2 Konfiguration

Zwecks Nachvollziehbarkeit und Wartbarkeit haben wir die ganze Cilium Konfiguration im File `cilium-values.yaml` vorgenommen:

`cilium-values.yaml`

```

1 # Der K3s Kube Proxy wurde deaktiviert, daher muss das Cilium Replacement aktiviert
  werden.
2 kubeProxyReplacement: true
3 # Der K3s API Hostname und Port, damit Cilium die Kubernetes API erreichen kann.
4 k8sServiceHost: k3s
5 k8sServicePort: 6443
6 # Hier wird der Adressbereich der Pods angegeben, welcher in K3s standardmässig
  verwendet wird.
7 ipam:
8   operator:
9     clusterPoolIPv4PodCIDRList: 10.42.0.0/16
10 # Diese Konfigurationen werden benötigt, damit hochverfügbare VIPs für Services
  funktionieren.
11 l2announcements:
12   enabled: true
13 externalIPs:
14   enabled: true

```

2.3.3 Installation

Mit Helm haben wir die neuste Version installiert:

```

1 helm install cilium cilium/cilium -n kube-system -f cilium-values.yaml

```

2.3.4 Konfiguration VIP

Der eingebaute Traefik Ingress Controller des K3s Controllers verfügt über einen Service mit dem Typ LoadBalancer. Dieser Service verlangt nach einer virtuellen IP Adresse, welche von ausserhalb des Clusters erreichbar ist. Über diese virtuelle IP Adresse kann dann hochverfügbar auf den Traefik Ingress Controller und somit auf alle über einen Ingress freigegebenen Applikationen im Cluster zugegriffen werden. Cilium bietet die Möglichkeit, einen IP Pool zu definieren, aus welchem diese virtuelle IP Adresse bezogen werden kann. Unser IP Pool besteht nur aus einer einzigen IP Adresse (10.0.10.10), da es nur einen LoadBalancer Service im Cluster gibt. Mit der L2 Announcement Policy stellen wir sicher, dass im Netzwerk auf ARP Requests für 10.0.10.10 mit der MAC Adresse des Nodes geantwortet wird, welches aktuell die virtuelle IP Adresse besitzt.

cilium-ip-pool.yaml

```

1 apiVersion: "cilium.io/v2alpha1"
2 kind: CiliumLoadBalancerIPPool
3 metadata:
4   name: "ingress-pool"
5 spec:
6   blocks:
7     - start: "10.0.10.10"
8       stop: "10.0.10.10"

```

cilium-announce.yaml

```

1 apiVersion: cilium.io/v2alpha1
2 kind: CiliumL2AnnouncementPolicy
3 metadata:
4   name: ingress-l2-announcement-policy
5   namespace: kube-system
6 spec:
7   externalIPs: true
8   loadBalancerIPs: true

```

```

1 kubectl apply -f cilium-ip-pool.yaml
2 kubectl apply -f cilium-announce.yaml

```

2.4 Etcd

Ein Patroni Cluster benötigt einen Distributed Consensus Store (DCS), um den Zustand des Postgres-Clusters zu speichern und zu verwalten. Der DCS fungiert als Quelle der Wahrheit für Patroni, um zu entscheiden, welche Aktionen durchgeführt werden sollen. Patroni unterstützt die DCS etcd, Consul und ZooKeeper.

Folgendes waren die Gründe für den Entscheid für etcd:

- Einfaches und starkes Konsistenz-Modell
- Simple Integration in Patroni
- Niedrige Komplexität bei Installation und Betrieb

2.4.1 Installation

Etcd ist in den offiziellen Ubuntu Repositories vorhanden, wir konnten es daher einfach in der Version 3.3 installieren, indem wir das folgende Kommando ausführten:

```

1 apt install etcd -y

```

2.4.2 Konfiguration

Die Konfigurationsdatei von etcd befindet sich unter /etc/default/etcd. Wir haben die folgenden Änderungen vorgenommen:

etcd

```

1 ETCD_LISTEN_PEER_URLS="http://<k3s-node-ip>:3380"
2 ETCD_LISTEN_CLIENT_URLS="http://<k3s-node-ip>:3379,http://127.0.0.1:3379"
3 ETCD_INITIAL_ADVERTISE_PEER_URLS="http://<k3s-node-ip>:3380"
4 ETCD_INITIAL_CLUSTER="k3s-01=http://10.0.10.11:3380,
5 k3s-02=http://10.0.10.12:3380, k3s-03=http://10.0.10.13:3380"
6 ETCD_INITIAL_CLUSTER_STATE="new"
7 ETCD_INITIAL_CLUSTER_TOKEN="supersecret"
8 ETCD_ADVERTISE_CLIENT_URLS="http://<k3s-node-ip>:3379"

```

Die benötigten Parameter haben wir aus der offiziellen Etcd Dokumentation für Cluster abgeleitet (<https://etcd.io/docs/v3.3/op-guide/clustering/#static>). Speziell ist, dass wir die Ports gegenüber den Standard-Ports um 1000 erhöht haben, weil die Standard-Ports bereits vom etcd in k3s verwendet wurden.

Nachdem wir die Konfiguration geändert haben, haben wir den etcd-Dienst neu gestartet, beim Booten aktiviert und die Firewall-Regeln angepasst:

```
1 systemctl restart etcd
2 systemctl enable etcd
3 ufw allow 3379:3380/tcp
```

2.5 PostgreSQL

Wir haben uns für die neuste PostgreSQL Version 16 entschieden, um die neusten Features und den längsten Support zu erhalten. In den Ubuntu 22.04 Repositories ist nur die Version 14 enthalten, wir haben PostgreSQL 16 daher von den offiziellen Repositories installiert gemäss offizieller Anleitung (<https://www.postgresql.org/download/linux/ubuntu/>). Ausserdem mussten wir die Firewall-Rules anpassen und den Service deaktivieren, da Patroni Postgres selbst ohne den Standard-Service verwaltet.

```
1 sh -c 'echo "deb https://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-pgdg
main" > /etc/apt/sources.list.d/pgdg.list'
2 wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-
key add -
3 apt-get update
4 apt-get -y install postgresql-16
5 systemctl disable --now postgresql.service
6 ufw allow 5432/tcp
```

2.6 Patroni

Wir verwenden Patroni als High-Availability Lösung für PostgreSQL, es verwaltet automatisch die Replikation und Failover von PostgreSQL. Wir haben uns dafür entschieden, weil es populär ist und wir bereits gute Erfahrungen damit gemacht haben.

2.6.1 Installation

Patroni ist in den offiziellen Ubuntu Repositories vorhanden, wir konnten es daher mit dem folgenden Befehl installieren:

```
1 apt -y install patroni
```

2.6.2 Konfiguration

Als Vorlage für die Konfiguration haben wir die Vorlage aus dem offiziellen Github von Patroni verwendet (<https://github.com/zalando/patroni/blob/master/postgres0.yml>). Diese haben wir in die Datei /etc/patroni/config.yml kopiert und angepasst. Folgendes wurde gegenüber der Vorlage geändert:

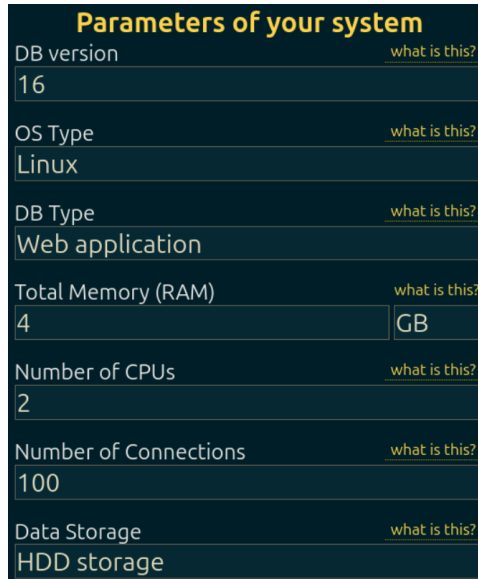
config.yml

```

1  scope: 16-sitepulse
2  name: k3s-03
3  restapi:
4    listen: <k3s-node-ip>:8008
5    connect_address: <k3s-node-ip>:8008
6  etcd3:
7    hosts:
8      - 10.0.10.11:3379
9      - 10.0.10.12:3379
10     - 10.0.10.13:3379
11 bootstrap:
12   dcs:
13     postgresql:
14       pg_hba:
15         - host replication replicator 10.0.10.0/24 scram-sha-256
16         - host all all 10.0.10.0/24 scram-sha-256
17       parameters:
18         max_connections: 100
19         shared_buffers: 1GB
20         effective_cache_size: 3GB
21         maintenance_work_mem: 256MB
22         checkpoint_completion_target: 0.9
23         wal_buffers: 16MB
24         default_statistics_target: 100
25         random_page_cost: 4
26         effective_io_concurrency: 2
27         work_mem: 5242kB
28         huge_pages: off
29         min_wal_size: 1GB
30         max_wal_size: 4GB
31         wal_level: hot_standby
32         hot_standby: "on"
33         max_worker_processes: 8
34         wal_keep_segments: 8
35         max_wal_senders: 10
36         max_replication_slots: 10
37         max_prepared_transactions: 0
38         max_locks_per_transaction: 64
39         wal_log_hints: "on"
40         track_commit_timestamp: "off"
41         archive_mode: "on"
42         archive_timeout: 1800s
43         archive_command: mkdir -p ../wal_archive && test ! -f ../wal_archive/%f
44         && cp %p ../wal_archive/%f
45       recovery_conf:
46         restore_command: cp ../wal_archive/%f %p
47     postgresql:
48       listen: <k3s-node-ip>:5432
49       connect_address: <k3s-node-ip>:5432
50       data_dir: /opt/postgres/sitepulse
51       bin_dir: /usr/lib/postgresql/16/bin
52       pgpass: /tmp/16-sitepulse.pgpass
53       authentication:
54         replication:
55           username: replicator
56           password: supersecret
57         superuser:
58           username: postgres
59           password: supersecret
60       rewind:
61         username: rewind_user
62         password: supersecret

```

Die IP-Adressen <k3s-node-ip> wurden durch die IP-Adresse des jeweiligen k3s-Nodes ersetzt. Die Passwörter wurden durch sichere Passwörter ersetzt. Bei den Parametern für die PostgreSQL-Konfiguration haben wir einerseits die auskommentierten, empfohlenen Werte aus der Vorlage übernommen und andererseits die vorgeschlagenen Werte von PGTune (<https://pgtune.leopard.in.ua/>) eingefügt. PGTune haben wir unserer Umgebung entsprechend auf folgende Werte konfiguriert:



Parameters of your system

DB version	what is this?
16	
OS Type	what is this?
Linux	
DB Type	what is this?
Web application	
Total Memory (RAM)	what is this?
4	GB
Number of CPUs	what is this?
2	
Number of Connections	what is this?
100	
Data Storage	what is this?
HDD storage	

Figure 5: PGTune Konfiguration

Danach haben wir die Berechtigungen gesetzt, die Firewall angepasst und Patroni gestartet:

```
1 chown postgres:postgres /opt/postgres
2 ufw allow 8008/tcp
3 systemctl enable --now patroni
```

Den Clusterstatus haben wir erfolgreich mit folgendem Befehl überprüft:

```
1 patronictl -c /etc/patroni/config.yml list
2
3 + Cluster: 16-sitepulse (7349220772731128835) ---+-----+
4 | Member | Host       | Role   | State   | TL | Lag in MB |
5 +-----+-----+-----+-----+---+-----+
6 | k3s-01 | 10.0.10.11 | Leader | running | 6  |           |
7 | k3s-02 | 10.0.10.12 | Replica | streaming | 6  |         0 |
8 | k3s-03 | 10.0.10.13 | Replica | streaming | 6  |         0 |
9 +-----+-----+-----+-----+---+-----+
```

2.7 HAProxy

Patroni benötigt einen Load Balancer, der die Anfragen an den derzeitigen Cluster-Leader weiterleitet. Wir haben uns für HAProxy entschieden, weil Patroni eine empfohlene HAProxy Konfiguration zur Verfügung stellt (<https://github.com/zalando/patroni/blob/master/haproxy.cfg>). HAProxy läuft containerisiert im K3s Cluster, weil es dort sehr einfach zu installieren ist und sowieso nur aus K3s auf die Datenbank zugegriffen wird. Die Kubernetes Manifeste befinden sich in unserem Gitlab Repository unter kubernetes/haproxy.

HAProxy fragt alle 3s mit http den Port 8008 (Patroni API) der drei Nodes an. Beim Leader kommt HTTP Code 200 zurück, bei den Followern HTTP Code 503. HAProxy leitet die Anfragen dann ausschliesslich an den Leader weiter. Write Operationen sind nur auf dem Leader erlaubt, Read Operationen

auf allen Nodes. Wenn der Leader ausfällt, wird der Follower mit der höchsten Priorität zum Leader. HAProxy erkennt das und leitet die Anfragen an den neuen Leader weiter.

HAProxy ist als DaemonSet in Kubernetes deployed. Ein DaemonSet stellt sicher, dass auf jedem Node im Cluster ein Pod läuft, somit ist HAProxy hochverfügbar und es gibt keine Single Point of Failure bei einem Node. Unsere Applikation greift auf den Service mit DNS-Namen `haproxy.haproxy.svc.cluster.local` zu. Cilium leitet Anfragen an diesen Service an einen HAProxy Pod weiter, welcher die Anfrage dann an den derzeitigen Patroni Leader weiterleitet.

2.8 GitLab Agent

Um unseren K3s Cluster an Gitlab anzubinden und somit `kubectl` Befehle in der Gitlab CI/CD Pipeline ausführen zu können, benötigen wir einen GitLab Agent. Der Gitlab Agent lässt sich als Helm Chart im Cluster installieren. Die Konfiguration des Agents kann im Gitlab Webinterface generiert werden.

Die Konfiguration aus dem Gitlab Webinterface haben wir in `gitlab-agent-values.yaml` abgelegt:

gitlab-agent-values.yaml

```
1 image:
2   tag: 'v16.9.1' # Version des GitLab Agents, entspricht der Gitlab Version
3
4 config:
5   token: 'glagent-supersecret' # In Gitlab generierter Token
6   kasAddress: 'wss://gitlab.diller.gl/-/kubernetes-agent/' # Adresse des Gitlabs
```

Dann haben wir den Gitlab Agent mit Helm installiert:

```
1 helm repo add gitlab https://charts.gitlab.io
2 helm repo update
3 helm install k3s-sitepulse gitlab/gitlab-agent -n gitlab-agent-k3s-sitepulse --
  create-namespace -f gitlab-agent-values.yaml
```

2.9 GitLab Runner

Um in Gitlab CI/CD Pipelines Images builden zu können, benötigen wir einen Gitlab Runner. Dieser wird auf dem K3s Cluster installiert und bei Gitlab registriert.

Die minimale Konfiguration haben wir in `gitlab-runner-values.yaml` abgelegt:

gitlab-runner-values.yaml

```
1 gitlabUrl: 'https://gitlab.diller.gl' # URL zu Gitlab
2 runnerToken: 'supersecret' # Token für die Registrierung des Runners bei Gitlab
```

Dann haben wir den Gitlab Runner mit Helm installiert:

```
1 helm repo add gitlab https://charts.gitlab.io
2 helm repo update
3 helm install gitlab-diller-runner gitlab/gitlab-runner -n gitlab-diller-runner --
  create-namespace -f gitlab-runner-values.yaml
```

Da wir Kubernetes YAML Manifeste in anderen Namespaces, als der Runner läuft, anwenden wollen, mussten wir noch die RBAC Berechtigungen anpassen. Der Runner startet jeweils einen zusätzlichen Concurrent Pods, um die `kubectl` Befehle auszuführen. Diese Concurrent Pods verwenden den Service Account default des Namespaces, in dem der Runner läuft. Es gibt leider keine Möglichkeit, einen anderen Service Account zu konfigurieren. Es ist security-technisch nicht optimal, den default Service Account mit weitgehenden Berechtigungen auszustatten, weil andere Pods im selben Namespace diese

Berechtigungen missbrauchen könnten. Da der Gitlab Runner aber der einzige Pod in diesem Namespace ist und wir darauf achten, dass das so bleibt, ist das Risiko vertretbar. Deshalb haben wir den Service Account default des Namespaces gitlab-diller-runner mit den nötigen Berechtigungen ausgestattet:

```
gitlab-runner-rbac.yaml
1 kind: ClusterRoleBinding
2 metadata:
3   name: gitlab-runner-cluster-wide-access
4 roleRef:
5   apiGroup: rbac.authorization.k8s.io
6   kind: ClusterRole
7   name: cluster-admin
8 subjects:
9 - kind: ServiceAccount
10  name: default
11  namespace: gitlab-diller-runner

1 kubectl apply -f gitlab-runner-rbac.yaml
```

2.10 Gitlab CI/CD

Gitlab Continuous Integration / Continuous Deployment (CI/CD) ermöglicht es uns, automatisch die Container Images für das Backend und Frontend zu bauen und ins K3s zu deployen. Dies stellt sicher, dass unsere Applikation sehr einfach, schnell und konsistent deployt werden kann. Die gesamte Konfiguration für Gitlab CI/CD ist in der Datei `.gitlab-ci.yml` im Root unseres Repos definiert.

Die Pipeline wird nur ausgeführt, wenn entweder die Umgebungsvariable `VERSION_BACKEND` oder `VERSION_FRONTEND` gesetzt ist (oder beide). Somit können Backend und Frontend individuell oder gemeinsam neu deployed werden. Man muss manuell im Gitlab UI die Umgebungsvariable(n) setzen und die Pipeline starten.

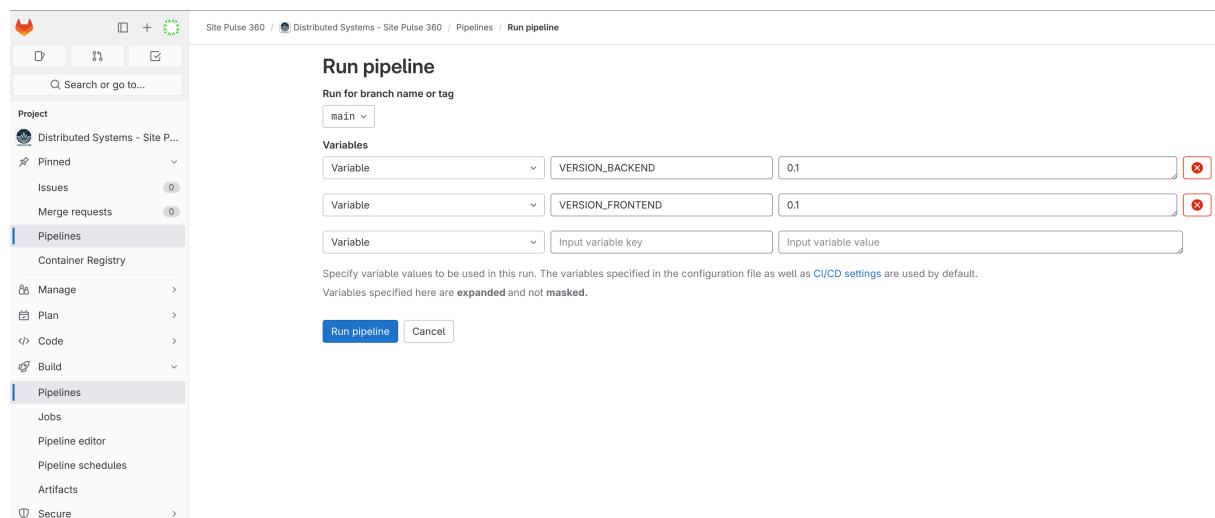


Figure 6: Pipeline starten

Die Images für das Backend und Frontend werden mit Kaniko gebaut und in die integrierte Gitlab Container Registry unter `registry.gitlab.diller.gl` gepusht. Kaniko benötigt keinen privileged Pod mit Docker-in-Docker, somit ist es die sicherste Methode, um Container Images in einer CI/CD Pipeline zu bauen.

Anschliessend werden die Kubernetes Ressourcen für das Backend, Frontend, HAProxy und Traefik in den Cluster deployed. Die Images werden dabei aus der Auth-geschützten Gitlab Container Registry gepullt, daher muss ein Pull Secret für den Zugriff auf das Registry erstellt werden. Die Umgebungsvariablen `CI_DEPLOY_USER` und `CI_DEPLOY_PASSWORD` mit dem Registry Login haben wir gesetzt, indem wir einen Deploy Token mit dem Namen `gitlab-deploy-token` erstellt haben. Der Token wurde nach dem Least Privilege Prinzip erstellt, er darf nur die Container Registry lesen und sonst nichts.

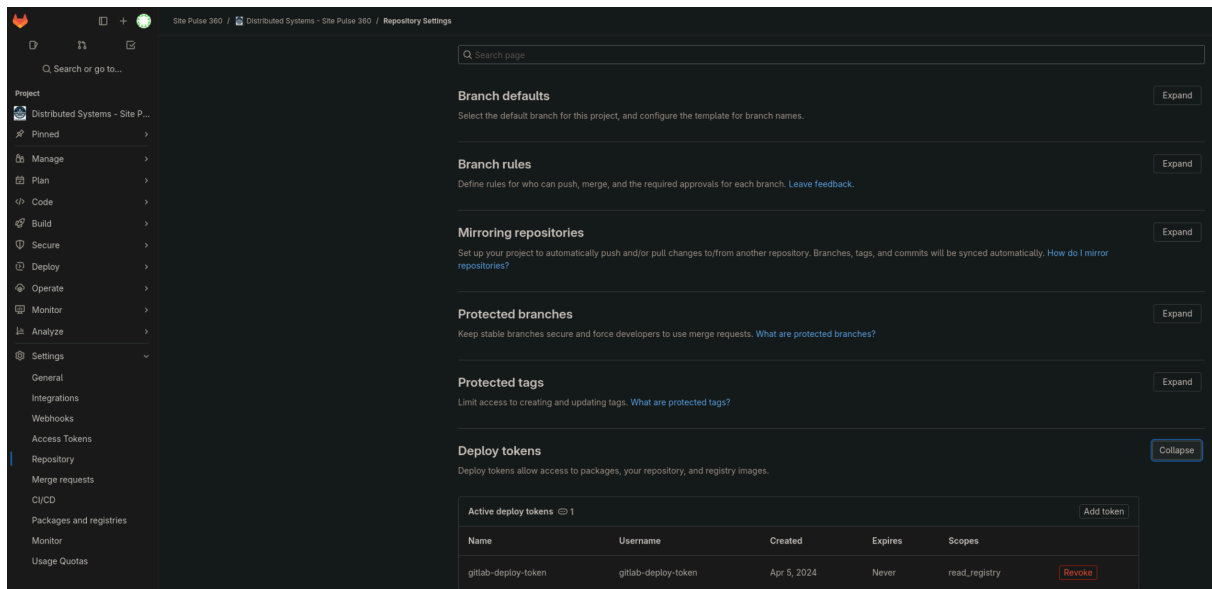


Figure 7: Deploy Token

3 Applikation

Die Applikation, die entwickelt wird, soll die Verfügbarkeit von hinterlegten Webseiten im gegebenen Intervall (jede Minute) überprüfen und anschliessend den Verlauf der letzten 30 Tage darstellen. Dieses Projekt wird in einen Frontend- und einen Backend-Teil aufgeteilt. Im Frontend sollen die Webseiten erfasst und angezeigt werden können. Für jede Webseite, die erstellt wird, wird diese im Backend hinterlegt und vom Backend werden anschliessend die Anfragen durchgeführt.

3.1 Frontend

Das Frontend wurde mithilfe von Vue.js umgesetzt. Vue.js ist ein JavaScript-Framework, welches auf der Client-Seite läuft und für die Erstellung von Single-Page-Applications (SPA) verwendet wird. Vue.js ist relativ einfach zu erlernen und bietet viele Möglichkeiten, um eine moderne Webanwendung zu erstellen. Vue.js bietet eine sehr gute Dokumentation und eine grosse Community, was die Entwicklung von Webanwendungen erleichtert.

Um das CSS Styling zu vereinfachen, wurde das CSS-Framework Bootstrap verwendet. Bootstrap ist ein Open-Source-Framework, welches viele vorgefertigte CSS-Klassen und -Komponenten bietet.

Das Frontend besteht aus einer Startseite mit einer Liste der überwachten Websites, einer Seite zur Erstellung von neuen zu überwachenden Websites und einer Seite, auf welcher Details zu einer überwachten Website angezeigt werden.

3.1.1 Startseite



Figure 8: Startseite mit Liste überwachter Websites

Auf der Startseite sieht man alle überwachten Websites. Die Liste zeigt den benutzerdefinierten Namen der Website, die URL und den aktuellen Status (up/down) der Website an. Durch den Klick auf die Kachel einer Website kommt man zur Detailseite und durch den Klick auf das + Symbol kommt man zur Erstellungsseite.

Diese Seite ist im File `HomeView.vue` implementiert. Bei jedem Laden wird die Liste und der Status vom Backend abgerufen und angezeigt. Pro Website wird ein Kachel-Element erstellt, welches in `WebsiteItem.vue` implementiert ist.

3.1.2 Neue Website erstellen

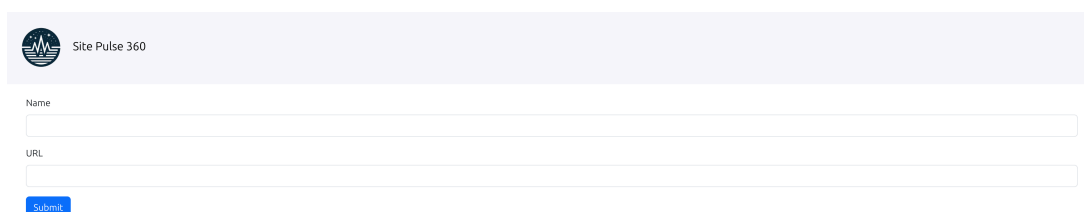


Figure 9: Neue zu überwachende Website erstellen

Auf dieser Seite kann durch Eingabe eines Namens und einer URL eine neue zu überwachende Website erstellt werden.

Das simple HTML Formular ist in `CreateView.vue` implementiert. Beim Absenden des Formulars wird ein POST-Request an das Backend gesendet, um die Website zu erstellen. Traefik verlangt dafür einen Basic Auth User und Passwort.

3.1.3 Website Details

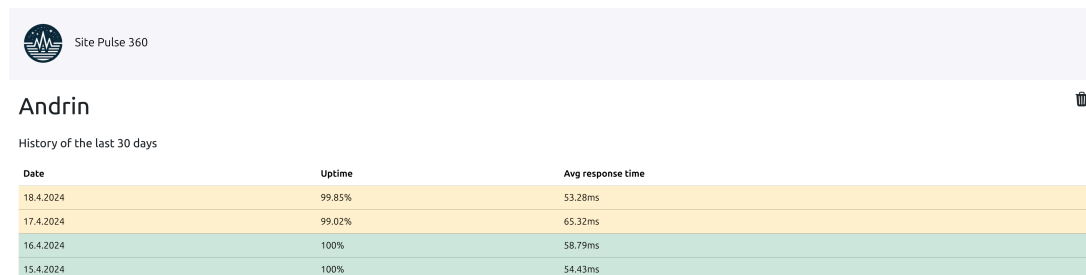


Figure 10: Details zu einer überwachten Website

Auf dieser Seite werden die Details einer überwachten Website angezeigt. Man sieht den Namen, eine History der letzten 30 Tage (oder weniger, falls die Website noch nicht so lange überwacht wird) und kann die Website löschen. In der History sieht man die prozentuale Verfügbarkeit und durchschnittliche Antwortzeit der Website aufgeteilt nach Kalendertagen. Je nach Verfügbarkeit wird die Zeile grün (100%), gelb (zwischen 99% und 100%) oder rot (unter 99%) eingefärbt.

Diese Seite ist im File `WebsiteView.vue` implementiert. Bei jedem Laden der Seite werden die letzten 43200 Monitoring-Einträge (30 Tage * 24h * 60 Minuten) vom Backend abgerufen. Danach werden die Daten nach Tag gruppiert und die prozentuale Verfügbarkeit und durchschnittliche Antwortzeit berechnet. Pro Tag wird dann eine Tabellenzeile erstellt, welche im Component `WebsiteDayItem.vue` implementiert ist.

Das Löschen einer Website wird durch einen DELETE-Request an das Backend durchgeführt. Traefik verlangt dafür einen Basic Auth User und Passwort.

3.1.4 Dockerfile

Beim Dockerfile haben wir uns für einen Multi-Stage-Build entschieden, dadurch erhalten wir ein viel kleineres Image und können den Vorteil von Vue.js, die ganze Applikation in den kleinen Ordner `dist` zu builden, optimal ausnutzen. In der ersten Stage wird zuerst das Node.js Image verwendet, um alle benötigten Node.js Modules zu installieren und die Applikation zu builden. In der zweiten und finalen Stage verwenden wir dann ein Nginx unprivileged Alpine Image, um das Image so klein wie möglich zu halten, aber doch noch gewisse Grundfunktionalitäten wie eine Shell zur Verfügung zu haben. Das unprivileged Image erhöht die Security, da der Hauptprozess nicht mit `root`, sondern mit dem eignen `nginx` User läuft. Wir kopieren den Inhalt des `dist` Ordners in den Nginx Webserver und konfigurieren den Webserver mit einer eigenen `nginx.conf` Datei. Zum Schluss wird der Port 8080 exposed und als Hauptprozess des Containers Nginx ohne Daemon gesetzt.

Dockerfile

```

1 # build stage
2 FROM node:22.2.0-alpine as build-stage
3 WORKDIR /app
4 COPY package*.json ./
5 RUN npm install
6 COPY . .
7 RUN npm run build
8
9 # production stage
10 FROM nginxinc/nginx-unprivileged:1.26.0-alpine as production-stage
11 COPY --from=build-stage /app/dist /usr/share/nginx/html
12 COPY nginx.conf /etc/nginx/conf.d/default.conf
13 EXPOSE 8080
14 CMD ["nginx", "-g", "daemon off;"]

```

3.2 Backend

Das Backend wurde in go geschrieben und hat folgende Funktionalitäten und Schnittstellen:

- API Endpoint für die Verwaltung von Webseiten und dessen Status
- URL-Scanner um den Status eines URLs ausfindig zu machen
- Leader election um ein Leader Node im Kubernetes Cluster zu bestimmen welches den Scanner jede Minute ausführt
- Datenbankanbindung an PostgreSQL für Webseiten- und Statusinformationen

Externe Bibliotheken:

- Aufgaben Planung:
 - github.com/go-co-op/gocron/v2
- Web Framework:
 - github.com/gofiber/fiber/v2
- ORM:
 - gorm.io/gorm
 - gorm.io/driver/postgres
 - gorm.io/gorm/logger
- Kubernetes:
 - k8s.io/apimachinery/pkg/apis/meta/v1
 - k8s.io/client-go/kubernetes
 - k8s.io/client-go/rest
 - k8s.io/client-go/tools/leaderelection
 - k8s.io/client-go/tools/leaderelection/resourcelock

3.2.1 API Endpoint

3.2.1.1 Ausgabe aller Webseiten

GET /api

Response

```

[
  {
    "ID": 49,
    "Name": "Andrin",
    "URL": "https://kessler.link/"
  },
  ...,
  {
    "ID": 82,

```

```
    "Name": "Gitlab",
    "URL": "https://gitlab.diller.gl/"
  }
]
```

3.2.1.2 Webseite erstellen

POST /api/website

Body

```
{
  "Name": "foo",
  "URL": "https://foo.bar/"
}
```

Response

Redirect zu "/"

3.2.1.3 Webseite löschen

DELETE /api/website/{websiteID}

Response

```
{
  "ID": <websiteID>,
  "Name": "foo",
  "URL": "https://foo.bar/"
}
```

3.2.1.4 Status einer Webseite abfragen

GET /api/website/{websiteID}/state

Response

```
[
  {
    "ID": 100,
    "CreatedAt": "2024-05-19T15:50:08.016902Z",
    "UpdatedAt": "2024-05-19T15:50:08.016902Z",
    "DeletedAt": null,
    "website": <websiteID>,
    "up": true,
    "return_code": 200,
    "request_date": "2024-05-19T15:50:07.773496Z",
    "response_time_ms": 242
  },
  ...
  {
    "ID": 1,
    "CreatedAt": "2024-05-18T15:49:08.033726Z",
    "UpdatedAt": "2024-05-18T15:49:08.033726Z",
    "DeletedAt": null,
    "website": <websiteID>,
    "up": true,
    "return_code": 200,
    "request_date": "2024-05-19T15:49:07.773956Z",
    "response_time_ms": 258
  }
]
```

3.2.2 URL-Scanner

Der Scanner liest alle URLs aus der Datenbank aus und führt für jeden URL einen asynchronen Scan aus. Der Scan versucht eine HTTP GET Anfrage auf den URL auszuführen. Kann eine HTTP Verbindung aufgebaut werden wird der HTTP Statuscode gespeichert ansonsten bleibt dieser leer. Eine Webseite/URL gilt als “up” sofern eine HTTP Verbindung aufgebaut werden konnte und dessen Statuscode kleiner als 400 ist.

HTTP Statuscode Übersicht:

- 100 Information
- 200 Erfolgreich
- 300 Weiterleitung
- 400 Client Fehler
- 500 Server Fehler

Mehr dazu unter: <https://httpwg.org/specs/rfc9110.html#status.codes>

3.2.3 Leader election

Durch die Leader election wird im Kubernetes Cluster ein Node ausgewählt das den Scanner ausführt. Mithilfe von gocron können Aufgaben in regelmässigen Intervallen ausgeführt werden. Somit kann gewährleistet werden, dass die URLs genau einmal pro Minute von nur einem Node gescannt werden.

Bei der Leader Election Auswahl wird eine Gruppe von Kandidaten bestimmt. Diese Kandidaten versuchen sich selber als Anführer zu deklarieren. Einer der Kandidaten gewinnt die Auswahl und wird zum Anführer. Der Anführer sendet kontinuierlich “heartbeats”, um seine Position als Anführer zu erneuern. Die anderen Kandidaten versuchen in regelmäßigen Abständen, Anführer zu werden. Dies stellt sicher, dass schnell ein neuer Anführer bestimmt wird, falls der aktuelle Anführer aus irgendeinem Grund ausfällt. Der Mechanismus nutzt das Kubernetes Koordinations Feature “Lease”. Der Lease Lock wird über die Kubernetes API angesprochen und bietet somit eine geteilte Ressource an mit welcher die Nodes den Leader ausfindig machen können.

Wichtigste Parameter:

- LeaseDuration (45s): Zeitspanne die ein Leader Kandidat abwarten muss bevor er die Leader Funktion versucht zu übernehmen
- RenewDeadline (40s): Zeitspanne in der der Leader versucht seine Führung zu erneuern bevor er aufgibt.
- RetryPeriod (2s): Pause zwischen Abfragen auf das Lease Objekt

Lease Objekt:

- name: leaderelection-scanner
- namespace: site-pulse-360
- acquireTime: Zeitpunkt zu dem der Lease erworben wurde
- holderIdentity: Identität vom Leader
- leaseDuration (45s): Zeitspanne die ein Leader Kandidat abwarten muss bevor er die Leader Funktion versucht zu übernehmen
- renewTime: Zeitpunkt an dem der Leader das Lease Objekt zuletzt aktualisiert hat

3.2.4 Datenbankanbindung

Liest, speichert und löscht Datensätze in der PostgreSQL Datenbank. Die Anbindung an die PostgreSQL Datenbank ist über eine gewöhnliche DSN konfiguriert. Für die Representation der Datenbankentitäten wurde die ORM Bibliothek GORM verwendet, welches die Übersetzung von go Structs und Datenbankentitäten übernimmt. Über Auto Migration wird das Schema jeweils bei App Start aktualisiert, wobei zu gunsten vom Schutz der Daten keine Spalten gelöscht werden. Die Zugriffe auf die Datenbank erfolgen bei GORM jeweils automatisch über Transaktionen um die Datenkonsistenz zu gewährleisten.

backend/models/models.go

```
type Website struct {
    gorm.Model
    Name string `json:"name" gorm:"index;not null;default:null"`
    URL string `json:"url" gorm:"index;not null;default:null"`
    States []Status `json:"states" gorm:"constraint:OnDelete:CASCADE"`
}

type Status struct {
    gorm.Model
    WebsiteID uint `json:"website" gorm:"not null;default:null"`
    Up bool `json:"up" gorm:"not null;default:false"` // on zero values the default
    gets used boolean false -> zero value
    ReturnCode uint `json:"return_code" gorm:"default:null"`
    RequestDate time.Time `json:"request_date" gorm:"index;not null;default:null"`
    ResponseTimeMs uint `json:"response_time_ms" gorm:"not null;default:null"`
}
```

3.2.5 Dockerfile

Beim Dockerfile haben wir uns für einen Multi-Stage-Build entschieden, dadurch erhalten wir ein viel kleineres Image und können den Vorteil von Golang, die ganze Applikation in ein einziges ausführbares Binary zu compilieren, optimal ausnutzen. In der ersten Stage werden zuerst alle benötigten Golang Modules installiert und danach die ganze Applikation ins File /usr/local/bin/sitepulse compiled. In der zweiten und finalen Stage verwenden wir dann ein Alpine Image, um das Image so klein wie möglich zu halten, aber doch noch gewisse Grundfunktionalitäten wie eine Shell zur Verfügung zu haben. Wir erstellen einen neuen User und Gruppe sitepulse und kopieren das Binary von der ersten Stage ins Image. Zum Schluss wird der Port 3000 exposed und als Hauptprozess des Containers das Binary gesetzt. Um die Security zu erhöhen, wird der Hauptprozess mit dem User sitepulse und nicht mit root ausgeführt.

Dockerfile

```
1 # build stage
2 FROM golang:1.22.3-alpine as build-stage
3 WORKDIR /usr/src/app
4 COPY go.mod go.sum ./
5 RUN go mod download && go mod verify
6 COPY . .
7 RUN go build -v -o /usr/local/bin/sitepulse ./cmd
8
9 # production stage
10 FROM alpine:3.19.1 as production-stage
11 RUN addgroup -g 1000 sitepulse && adduser -h /home/sitepulse -s /bin/sh -G
12   sitepulse -D -u 1000 sitepulse
13 USER 1000:1000
14 COPY --from=build-stage --chown=1000:1000 --chmod=0770 /usr/local/bin/sitepulse /
15   usr/local/bin/sitepulse
16 EXPOSE 3000
17 CMD ["sitepulse"]
```

3.3 Ausführen der Applikation

3.3.1 Voraussetzungen

Das Ausführen der Applikation hat folgende Voraussetzungen:

- Container Registry (fürs Pushen und Pullen der Frontend und Backend Images)
- Kubernetes Cluster (fürs Deployen der Frontend und Backend Pods)
- Traefik Ingress Controller im Kubernetes Cluster (fürs Routen der Anfragen an die richtigen Services und Basic Auth)
- PostgreSQL Datenbank (fürs Speichern der Daten)
- Bash mit docker (eingeloggt auf Container Registry) und kubectl (konfiguriert auf Kubernetes Cluster) installiert

3.3.2 Deployment

Um die Applikation zu deployen, müssen folgende Schritte durchgeführt werden:

1. Images bauen mithilfe der Dockerfiles im frontend und backend Verzeichnis und in die Container Registry pushen.

```
1 export CI_REGISTRY_IMAGE= # z.B. registry.example.com/repository
2 export VERSION_BACKEND= # z.B. 1.0
3 export VERSION_FRONTEND= # z.B. 1.0
4 cd backend/
5 docker build -t ${CI_REGISTRY_IMAGE}/backend:${VERSION_BACKEND} .
6 docker push ${CI_REGISTRY_IMAGE}/backend:${VERSION_BACKEND}
7 cd ../frontend/
8 docker build -t ${CI_REGISTRY_IMAGE}/frontend:${VERSION_FRONTEND} .
9 docker push ${CI_REGISTRY_IMAGE}/frontend:${VERSION_FRONTEND}
```

2. Deployment Umgebungsvariablen setzen

```
1 export BASIC_AUTH_PASSWORD= # Basic Auth Passwort bei POST/DELETE Requests an API
2 export BASIC_AUTH_USERNAME= # Basic Auth User bei POST/DELETE Requests an API
3 export CI_DEPLOY_PASSWORD= # Passwort fürs Pullen der Images
4 export CI_DEPLOY_USER= # User fürs Pullen der Images
5 export CI_REGISTRY= # Registry fürs Pullen der Images, z.B. registry.example.com
6 export DB_HOST= # Hostname/IP der PostgreSQL Datenbank
7 export DB_NAME= # Name der PostgreSQL Datenbank
8 export DB_PASSWORD= # Passwort der PostgreSQL Datenbank
9 export DB_PORT= # Port der PostgreSQL Datenbank
10 export DB_USER= # Benutzername der PostgreSQL Datenbank
11 export HOST= # Externer Hostname der Site Pulse 360 Applikation
```

3. Kubernetes Deployment ausführen

```
1 kubectl create ns site-pulse-360
2 kubectl create secret docker-registry registry-gitlab-diller --docker-
server ${CI_REGISTRY} --docker-username ${CI_DEPLOY_USER} --docker-password
${CI_DEPLOY_PASSWORD} -n site-pulse-360
3 for folder in kubernetes/*; do cat $folder/* | envsubst | kubectl apply -f -; done
```

4 Fazit

Grundsätzlich können wir sagen, dass unser Projekt erfolgreich verlaufen ist. Die Planung und Koordinierung im Team hat immer gut funktioniert und für aufgetretene Probleme wurde immer eine Lösung gefunden. Die Applikation ist aus dem Internet erreichbar und erfüllt ihre Aufgabe.

Dieser Challenge Task hat uns einen Einblick in neue Technologien gegeben, darunter Kubernetes, GitLab CI/CD, Patroni, Golang und Vue.js. Während die meisten von uns in diesen Bereichen die ersten Schritte gemacht haben, konnten wir aber auch in vertrauten Gebieten weitere Erfahrungen sammeln.

Während der Umsetzung dieses Projekts wurden wir mit manchen Problemen konfrontiert. Da der Kubernetes-Cluster in einer produktiven Infrastruktur eingerichtet wurde, waren bereits diverse Dienste gehostet. Die Zugriffe über HTTPS werden durch einen NGINX Reverse-Proxy geregelt, welcher aufgrund des DNS-Namen im Request die Anfrage an die richtige Ressource weiterleitet. Um dennoch die Konfiguration des Kubernetes-Clusters und der benötigten HA IPs und Load-Balancer wie im produktiven Umfeld darzustellen, wurde nicht auf die restlichen Funktionalitäten von NGINX zurückgegriffen.

Eine andere Frage, die wir uns im Laufe der Umsetzung stellen mussten, war, ob wir die API ins Internet stellen, damit der User die Anfrage an die API sendet, oder ob alles Cluster-intern geschieht. Zweiteres würde aber bedeuten, dass die Applikation komplett auf dem Server gerendert werden muss. Da wir bereits an der Umsetzung des Frontends dran waren und die Umstellung in eine Server-Side Rendered Applikation einen deutlichen Mehraufwand bedeutet hätte, haben wir die API öffentlich gemacht. Dies hatte dann zur Folge, dass wir eine Form von Authentifizierung ins Backend einbauen mussten. Letzten Endes konnten wir aber auch diese Hürde meistern mithilfe von Traefik Basic Auth.

Zu Beginn haben wir die High-Availability IP des K3s-Clusters über MetalLB und das Kubernetes-interne Netzwerk über Cilium bereitgestellt, weil uns nicht bewusst war, dass Cilium die Funktionalität von MetalLB übernehmen kann. Um die Komplexität etwas zu verringern, haben wir gegen Ende die HA IP ebenfalls über Cilium bereitgestellt, wodurch MetalLB überflüssig wurde.

Wir sind mit unserer finalen Infrastruktur, Applikation und der generellen Umsetzung definitiv zufrieden.

A Abbildungsverzeichnis

Figure 1: Visualisierung Techstack + Aufbau	1
Figure 2: Netzwerkplan Sitepulse Netzwerk	2
Figure 3: K3s Cluster Aufbau	2
Figure 4: Ubuntu Disk Layout	3
Figure 5: PGTune Konfiguration	10
Figure 6: Pipeline starten	12
Figure 7: Deploy Token	13
Figure 8: Startseite mit Liste überwachter Websites	14
Figure 9: Neue zu überwachende Website erstellen	14
Figure 10: Details zu einer überwachten Website	15