GitLab (sicher) selbst Hosten

Ausgangssituation: GitLab soll für die Verwaltung der eigenen Software Projekte im Unternehmen selbst gehostet werden. Dies bringt den Vorteil, dass die Daten in-house liegen können und nicht bei einem externen (Cloud-)Anbieter.

Im folgenden wollen wir gemeinsam GitLab auf einem Linux Server (Debian 11) installieren und konfigurieren, sowie Punkte wie Backup und mögliche Absicherungen (Firewall, Fail2Ban, ...) besprechen und einrichten.

Wir starten mit einer frisch installierten Debian Instanz (*Debian 11.6, Stand März 2023*), neben einem OpenSSH-Server und den vorinstallierten Tools wurde noch nichts eingerichtet.

Alle Schritte am Debian-Server werden via Kommandozeile durchgeführt. Dazu wird vorrangig der **root**-Benutzer verwendet. Welche Nachteile dies haben kann, wird im laufe des Workshops besprochen.

Eckdaten

- Debian 11.6 net-install
- Benutzer:
  - *root*: *ke1nes*
  - *user*: *ke1nes*

> Anmerkung: Es kann grundsätzlich jede beliebige Linux-Distribution verwendet werden.

> **Benutzer**: Wir erledigen alle Schritte im diesem Workshop mit dem **root**-Benutzer. Alternativ sind die meisten Befehle ansonsten mit `sudo` auszuführen.

VirtualBox

- 2 Netzwerkadapter
  - NAT (für Internetzugriff von VM)
  - Host-Only (für SSH Zugriff auf VM)
- 2 CPU
- 4GB RAM

Installationsmöglichkeiten

GitLab steht als Omnibus-bundle für diverse Linux-Plattformen als in Form eines Repositories bereit. In diesem Workshop verwenden wir Docker als Ziel-Plattform.

GitLab - Unterschiedliche Versionen

GitLab steht in unterschiedlichen Versionen zur Verfügung. In diesem Workshop nutzen wir die frei nutzbare Community-Edition (CE), welche alle Basisfunktionen zur Verfügung stellt.
Es kann jederzeit Problemlos von der CE auf die Enterprise-Edition (EE) umgestellt werden, die vorhanden Daten (Repositories, Projekte, etc.) können übernommen werden.

Docker Installation

Docker Installation unter Debian.

```
apt update
apt install ca-certificates curl gnupg lsb-release
mkdir -m 0755 -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o
/etc/apt/keyrings/docker.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/debian
$(lsb_release -cs) stable" | tee /etc/apt/sources.list.d/docker.list >
/dev/null
```

Als erstes fügen wir das Repository von Docker unserem System hinzu.

Nun sollten wir die aktuellste Version von Docker installieren können.

```
apt update
apt install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
docker-compose-plugin
```

Zum Verifizieren, dass Docker läuft, ein erstes Image starten.

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest:
sha256:6e8b6f026e0b9c419ea0fd02d3905dd0952ad1feea67543f525c73a0a790fefb
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs
the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent
it
    to your terminal.
...
```

Docker läuft als Systemdienst. Zum überprüfen ob der Daemon läuft kann `systemctl` genutzt werden.

```
$ systemctl status docker.service
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor
preset: enabled)
     Active: active (running) since Mon 2023-03-13 12:08:57 GMT; 1min 34s
ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 2294 (dockerd)
      Tasks: 9
     Memory: 36.5M
        CPU: 356ms
     CGroup: /system.slice/docker.service
             └─2294 /usr/bin/dockerd -H fd:// --
containerd=/run/containerd/containerd.sock
```

Mittels dem Befehl `systemctl` können die Services verwaltet werden.

- `systemctl start <service>`: Startet gewünschtes Service
- `systemctl restart <service>`: Startet Service neu durch
- `systemctl reload <service>`: Lädt Konfigurationsänderungen neu
- `systemctl stop <service>`: Stopt das Service
- `systemctl status <service>`: Gibt den aktuellen Status des Service aus
- `journalctl -u <service> [-f]`: Gibt das Log des Service aus
- `systemctl enable/disable <service>`:
  - `enable`: ermöglicht den "Autostart" des Service bei Systemstart
  - `disable`: deaktiviert den "Autostart" des Service bei Systemstart

Sofern Docker als Systemdienst läuft, sind root-Rechte nötig um Container zu starten/verwalten. "Normale" Benutzer benötigen `sudo`-Rechte oder sind der `docker` Gruppe hinzuzufügen (**Achtung**: gibt dem Benutzer indirekt Root-Rechte!).

```
usermod -aG docker <user>
```

Alternative: Docker Rootless

GitLab Container starten

Nachdem wir erfolgreich Docker auf unserem System installiert haben, können wir nun GitLab direkt starten. GitLab steht als Docker-Image mit allen nötigen Abhängigkeiten bereit, wodurch wir direkt einen Container starten können.

```
docker run --detach \
  --hostname gitlab.dih.at \
  --publish 80:80 --publish 443:443 \
  --volume $PWD/gitlab/config:/etc/gitlab \
  --volume $PWD/gitlab/logs:/var/log/gitlab \
  --volume $PWD/gitlab/data:/var/opt/gitlab \
  --shm-size 256m \
  --restart always \
  --name gitlab \
  gitlab/gitlab-ce:15.8.4-ce.0
```

- `--detach`: Started Container im Hintergrund
- `--hostname gitlab.dih.at`: setzt den Hostnamen des Containers. Damit können wir einfach via Hostnamen zugreifen
- `--publish HOST:CONTAINER`: Bindet den *CONTAINER*-Port and den *HOST*-Port.
- `--volume`: Bindet das lokale Directory im Container ein. Ermöglicht den lokalen Zugriff auf die Daten.
- `--shm-size 256m`: Definiert die Größe von `dev/shm` (*Shared Memory*) im Container. Von GitLab empfohlen auf 256 MB zu setzen.
- `--restart always`: Gibt an ob/wie der Container gestartet werden soll (zB wenn nachdem der Host durchgestartet wurde). `always` startet den Container selbstständig, nachdem `docker.service` gestartet wurde.
- `--name gitlab`: Gibt dem Container einen Namen, wodurch der Zugriff mittels CLI erleichtert wird. Name muss eindeutig sein!
- `gitlab/gitlab-ce:15.8.4-ce.0`: Als letztes Attribut gibt man das gewünschte Image an, aus dem ein Container gestartet werden soll. In dem Fall die Community Edition in der Version `15.8.4` *wir verwenden aktuell nicht `latest` um später den Update-Prozess durchzugehen*

Beim Start des Containers führt GitLab die komplette Initialisierung selbstständig durch. Dies kann einige Minuten in Anspruch nehmen (vor allem beim ersten Start). Den Fortschritt bzw. generell das Log kann mit folgenden Befehl verfolgt werden.

```
docker container logs -f gitlab
```

- `-f`: Für **follow**, neue Log-Nachrichten werden direkt angezeigt (`STRG+C` zum beenden)
- *gitlab*: Name des Containers (beim ausführen `docker run ... --name gitlab`) mit der Option `--name <NAME>` gewählte Name

> `docker container ls` zum anzeigen der erstellten Container.
> `docker container ls --all` zeigt auch gerade nicht laufenden Container an

In der Zwischenzeit können wir auf unserem Host-System einen *hosts*-Eintrag vornehmen, damit wir unsere GitLab Installation im Browser aufrufen können.

- **Windows:** `C:\Windows\System32\drivers\etc\hosts`
- **Linux/macOS:** `/etc/hosts`

ACHTUNG: Die `hosts` Datei muss mit Administrator-Rechten geöffnet werden!

```
# <IP> <hostname>
192.168.56.101   gitlab.dih.at
```

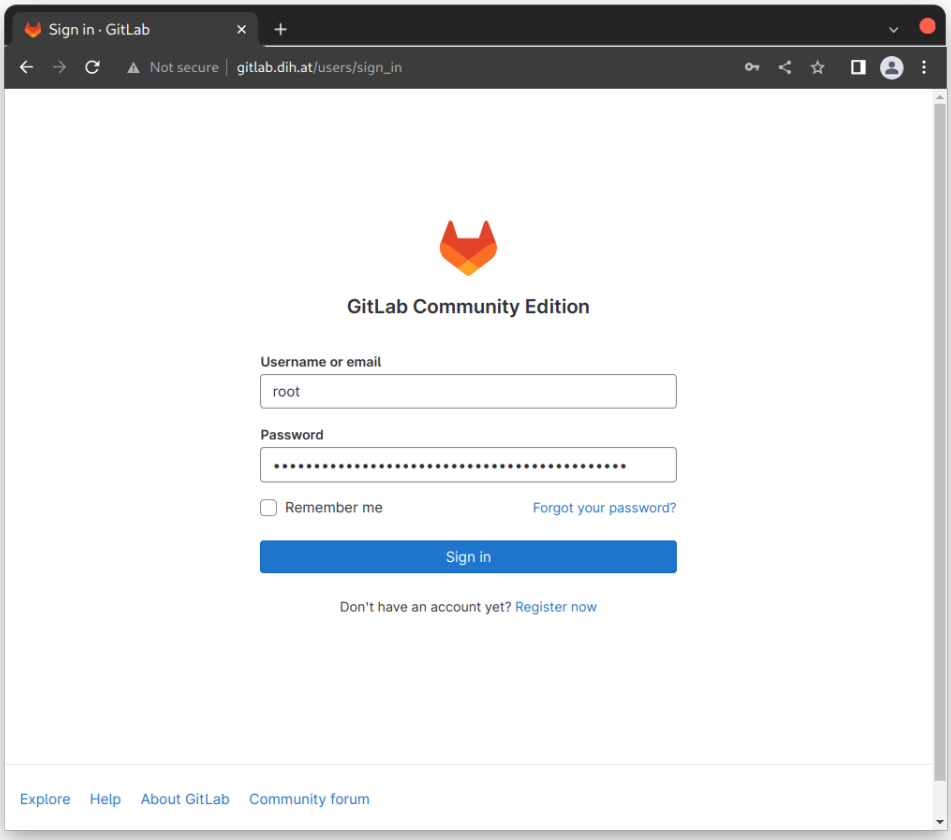
> Achtung: IP-Adresse kann und hostname können abweichen!

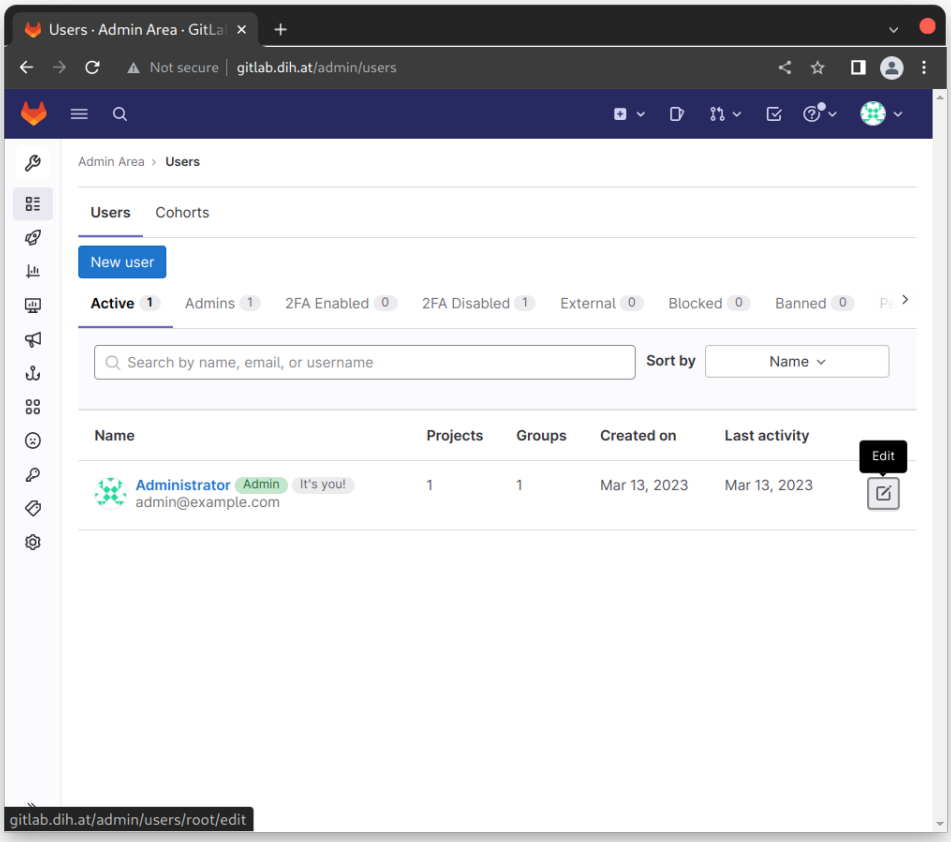
> Um die IP-Adresse der VM zu erfahren, in ihr den folgenden Befehl ausführen
>
> ```
> ip a
> ```
>
> Es wird eine Liste mit den vorhandenen Netzwerk-Interfaces aufgeführt. Das VirtualBox Image sollte über 2 Ethernet-Schnittstellen mit Namen `enp0s3` und `enp0s8` verfügen. `enp0s8` müsste das Host-Only-Interface sein und eine IP-Adresse wie zB `192.168.56.101` erhalten haben.

Bei der Initialisierung wird ein Standardbenutzer **root** mit einem Random-Passwort eingerichtet. Das initiale Passwort befindet sich in `/etc/gitlab/initial_root_password`.

```
cat $PWD/gitlab/config/initial_root_password
```

```
docker exec gitlab cat /etc/gitlab/initial_root_password
```





Unter `/admin/users` das Passwort und den Benutzernamen des "Administrators" ändern!

Den Container jedes mal "per Hand" zu starten ist ein wenig unflexibel, vor allem da einige Optionen anzugeben sind - und auch noch einige dazukommen.

Aus diesem Grund wechseln wir auf `docker compose` und legen die Optionen für das Service in einer YAML-Datei fest.

docker-compose.yml

```
version: '3.9'
services:
  gitlab:
    image: 'gitlab/gitlab-ce:15.8.4-ce.0'
    restart: always
    hostname: 'gitlab.dih.at'
    environment:
      GITLAB_OMNIBUS_CONFIG: |
        external_url 'http://gitlab.dih.at'
    ports:
      - '80:80'
      - '443:443'
    volumes:
      - '$GITLAB_HOME/config:/etc/gitlab'
      - '$GITLAB_HOME/logs:/var/log/gitlab'
      - '$GITLAB_HOME/data:/var/opt/gitlab'
    shm_size: '256m'
```

Damit die Pfade korrekt übernommen werden, müssen wir noch `$GITLAB_HOME` als Environment-Variable festlegen.

Dies machen wir am besten in der `~/.bashrc`, damit diese bei jedem Login korrekt gesetzt wird.

~/.bashrc

```
# ~/.bashrc
# ...

export GITLAB_HOME=$HOME/gitlab
```

Danach müssen wir die Konfiguration noch einmal einlesen.

```
source ~/.bashrc
```

Zum Verifizieren, dass die Variable korrekt gesetzt ist, können wir diese einfach mittels `echo` ausgeben, oder mittels `env` (oder auch `export`) alle Environment-Variablen ausgeben lassen.

```
echo $GITLAB_HOME
env | grep GITLAB_HOME
```

Bevor wir GitLab nun via `docker compose` starten, beenden wir erst einmal den aktuell laufenden Container.

```
docker container stop gitlab && docker container rm gitlab
```

Und starten ihn neu, nur dieses mal mittels `docker compose`.

> Dadurch, dass wir dieselben Volumes nutzen sind die bisherigen Einstellungen (Admin-User) persistent.

```
docker compose up -d
```

Docker-Compose bietet die meisten Sub-Befehle wie Docker selbst, zB `logs`.

```
docker compose logs -f
```

Ordner und Wichtige Dateien

Wir haben drei Locations aus dem Container ins lokale Dateisystem gemountet. Dort befinden sich die zentralen Daten für GitLab.

- `$DOCKER_HOME/config`: Konfigurationsdateien und Secrets, wie verwendete Keys für SSH. Im Container zu finden unter `/etc/gitlab`. Grundsätzlich wird GitLab über die `/etc/gitlab/gitlab.rb` Konfiguriert. Da wir Docker-Compose verwenden, werden wir alle nötigen Konfigurationen direkt in der Compose-Definition vornehmen. **Wichtig** hier ist vor allem die `gitlab-secrets.json`. Darin befinden sich die Keys, die zB für die Verschlüsselung in der Datenbank verwendet werden. Diese darf nicht abhanden kommen und muss vor allem bei einem Backup für einen späteren erfolgreichen Restore gesichert werden!
- `$DOCKER_HOME/logs`: Sämtliche Logdateien, die von GitLab produziert werden. Unter `$DOCKER_HOME/logs/gitlab-rails/` finden sich die wichtigsten Logs wenn es um Troubleshooting geht, wie `application.log` und `production.log`. Alle Webserver-spezifischen Logs finden sich unter `$DOCKER_HOME/logs/nginx`, SSH relevante unter `$DOCKER_HOME/logs/sshd`. Im Container finden sich die Logs unter `/var/log/gitlab`
- `$DOCKER_HOME/data`: Die Daten, die von GitLab produziert werden. Die Repositories findet man zB unter `$DOCKER_HOME/data/git-data`. Bzw. die Backups (dazu später mehr) unter `$DOCKER_HOME/data/backups`. Im Container ist das Verzeichnis unter `/var/opt/gitlab` zu finden.

GitLab nutzt innerhalb vom Container nicht ausschließlich den root-User, sondern Context-spezifisch unterschiedliche Benutzer.

```
$ docker compose exec gitlab cat /etc/passwd
...
git:x:998:998::/var/opt/gitlab:/bin/sh
gitlab-www:x:999:999::/var/opt/gitlab/nginx:/bin/false
gitlab-redis:x:997:997::/var/opt/gitlab/redis:/bin/false
gitlab-psql:x:996:996::/var/opt/gitlab/postgresql:/bin/sh
mattermost:x:994:994::/var/opt/gitlab/mattermost:/bin/sh
registry:x:993:993::/var/opt/gitlab/registry:/bin/sh
gitlab-prometheus:x:992:992::/var/opt/gitlab/prometheus:/bin/sh
gitlab-consul:x:991:991::/var/opt/gitlab/consul:/bin/sh
```

Entsprechend ist zum Durchsehen der Verzeichnisse am Host-System grundsätzlich der Root-User notwendig, auch wenn man einen anderen Benutzer für die GitLab Verwaltung nutzen würde! Alternativ bietet sich ansonsten an, alle Administrativen tätigen mittels `docker compose exec` durchzuführen, bzw. Daten zwischen Container und Host mit `docker compose cp` zu transferieren.

GitLab SSH-Zugriff

Um auf in GitLab verwaltete Remote-Repositories zuzugreifen gibt es 2 Varianten. SSH und HTTP(S). Die HTTP(S) Variante ist direkt verfügbar, da wir bereits die Ports 80 und 443 am Host gebunden haben. Was jetzt noch fehlt ist der SSH Port.

Da wir am Host selbst bereits Port 22 für SSH nutzen, gibt es 2 Varianten, wie wir SSH in GitLab erhalten.

1. Wir ändern den SSH Port des Hosts und nutzen Port 22 für GitLab
2. Wir nutzen einen alternativen Port für SSH am Host und binden diesen an Port 22 des Containers

In diesem Workshop wollen wir Variante 2 nutzen. Dafür müssen wir die Docker-Compose Konfiguration anpassen.

docker-compose.yml

```
# ...
    environment:
      GITLAB_OMNIBUS_CONFIG: |
        # ...
        gitlab_rails['gitlab_shell_ssh_port'] = 2222
    ports:
      - '80:80'
      - '443:443'
      - '2222:22'
# ...
```

Wir fügen das Binding für den SSH Port hinzu. In diesem Beispiel verwenden wir Port 2222 am Host um binden ihn an Port 22 des Containers. Damit GitLab intern die Links korrekt anzeigt und funktioniert, müssen

wir die *GITLAB_OMNIBUS_CONFIG* entsprechend erweitern.

Um die Änderungen zu übernehmen starten wir die Compose Service neu.

```
docker compose up -d
```

Der Container wird mit den neuen Einstellungen neu erstellt/gestartet. Wir können in der Wartezeit verifizieren, dass das Port-Bindung funktioniert.

```
$ ss -tlpn
State           Recv-Q          Send-Q                      Local
Address:Port                     Peer Address:Port          Process
LISTEN          0               4096
0.0.0.0:2222                          0.0.0.0:*              users:
(("docker-proxy",pid=4783,fd=4))
LISTEN          0               4096                              0.0.0.0:80
0.0.0.0:*              users:(("docker-proxy",pid=4763,fd=4))
LISTEN          0               128                               0.0.0.0:22
0.0.0.0:*              users:(("sshd",pid=420,fd=3))
LISTEN          0               4096
0.0.0.0:443                           0.0.0.0:*              users:
(("docker-proxy",pid=4744,fd=4))
```

`ss` ist ein Tool zum überprüfen der genutzten Sockets.

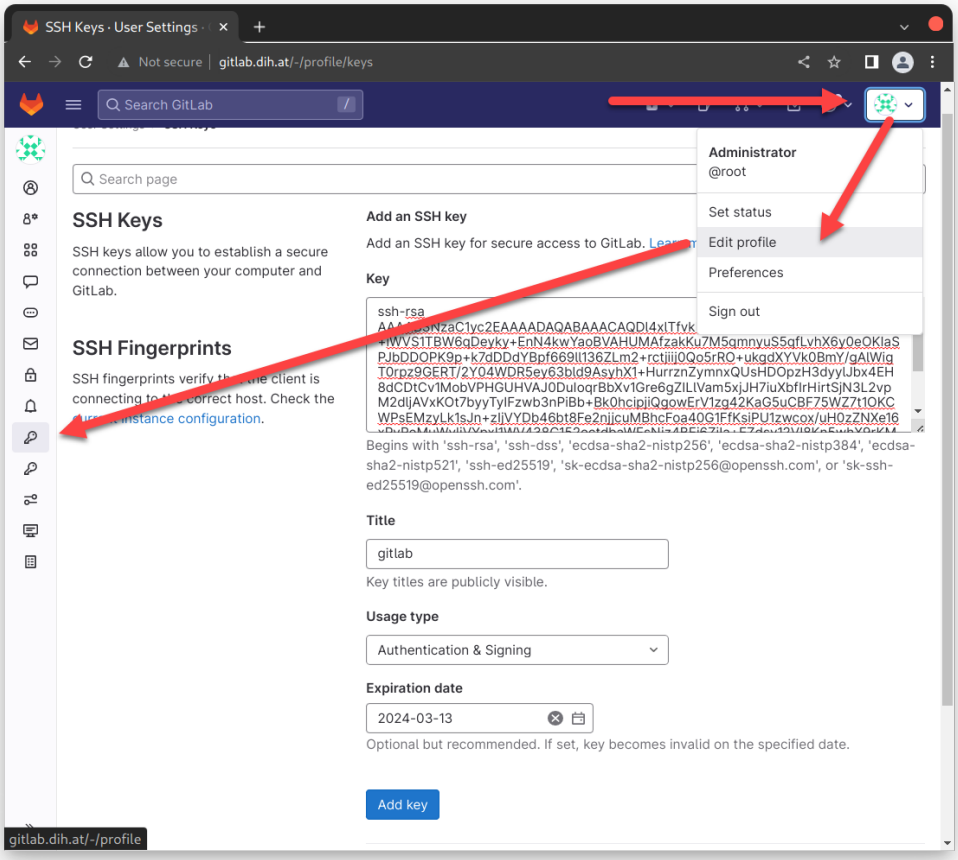
Wir sollten Port 2222 in der Auflistung finden, neben 80 (HTTP) und 443 (HTTPS), sowie 22 (SSH), der direkt vom OpenSSH-Server am Host genutzt wird.

Sobald der Zugriff auf die Website wieder möglich ist, können wir den Remote-Zugriff mittels `git` ausprobieren. Dafür erstellen wir einmal ein SSH-Keypair.

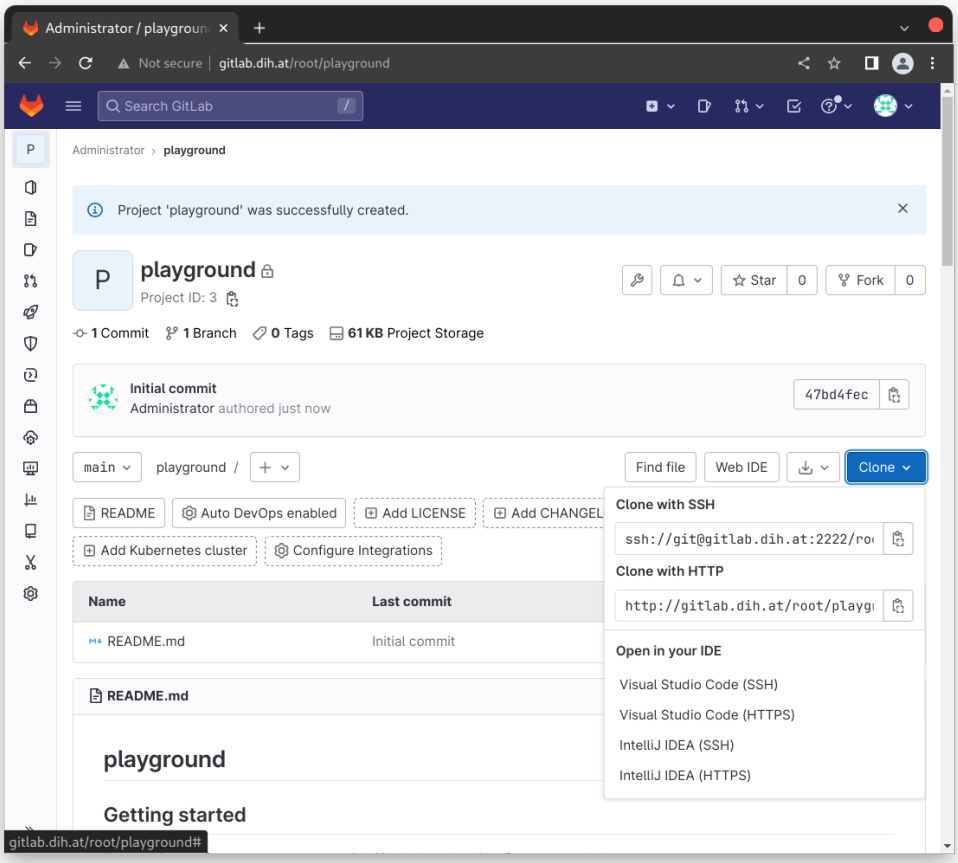
```
ssh-keygen -t rsa -b 4096 -C gitlab
```

Den Public-Key hinterlegen wir dann bei unserem Benutzerprofil auf GitLab.

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa .... gitlab
```



Public-Key unter `/-/profile/keys` hinzufügen.



`git clone via ssh oder http(s)`

Wir sollten nun in der Lage sein, ein Repository mittels SSH und/oder HTTP(s) zu clonen/pullen/pushen.

```
$ git clone ssh://git@gitlab.dih.at:2222/root/playground.git playground_ssh
Cloning into 'playground_ssh'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
$ git clone http://gitlab.dih.at/root/playground.git playground_http
Cloning into 'playground_http'...
Username for 'http://gitlab.dih.at': root
Password for 'http://root@gitlab.dih.at':
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

GitLab Update

GitLab bringt regelmässig Updates heraus. Es empfiehlt sich diese zeitlich einzuspielen. Wird GitLab über das Linux-Repository installiert, geschieht das Update einfach über den normalen Update Prozess (`apt upgrade` zB). Über Docker müssen wir einfach nur das aktuelle (bzw. gewünschte) Image laden und den Container neu starten.

Dafür passen wir einfach in der `docker-compose.yml` die Version an. Nutzen wir gleich `:latest` müssen wir hier auch nicht manuell die Version anpassen.

docker-compose.yml

```
# ...
    image: 'gitlab/gitlab-ce:latest'
# ...
```

Nun das gewünschte Image von docker.io pullen und Service neu starten.

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

Gibt es einen Sprung in der Major Version, kann es sein, dass ein entsprechender Update-Pfad einehalten werden muss, sollte nicht regelmäßig auf die letzte Version upgedated werden!

Backup und Restore

Für das erstellen von Backups bietet GitLab selbst ein Commandline-Tool an.

```
docker compose exec -t gitlab gitlab-backup
```

Das Backup befindet sich dann in Form eines tar-Archivs unter `$GITLAB_HOME/data/backups`.

ACHTUNG: Es werden dabei nicht die secrets (`$GITLAB_HOME/config/gitlab-secrets.json` etc.) mitgesichert! Diese sind wenn nötig separat zu sichern. Für ein vollständiges Restore (zB auf einer anderen Maschine) sind diese notwendig!

```
docker compose exec -t gitlab /bin/sh -c 'gitlab-ctl backup-etc && cd
/etc/gitlab/config_backup && cp $(ls -t | head -n1)
/var/opt/gitlab/backups/'
```

Die Backup-Archive können dann entweder direkt vom Host-Share (`$GITLAB_HOME/data/backups`) wegkopiert werden, bzw aus dem Container kopiert und an eine entsprechendes Backup-Storage übertragen werden.

```
docker compose cp gitlab:/var/opt/backups/<BACKUP>.tar .
```

Restore

Das Wiederherstellen eines Backups funktioniert ebenfalls mit dem Kommandozeilen-Tool von GitLab. Das Backup-Archiv muss dafür in `$GITLAB_HOME/data/backups` (`/var/opt/gitlab/backups`) liegen (inkl.passender Rechte -> Benutzer `git` (uid=998)).

Für die Wiederherstellung sollten die Services `puma` und `sidekiq` nicht laufen, da diese mit der Datenbank kommunizieren.

```
docker compose exec gitlab gitlab-ctl stop puma
docker compose exec gitlab gitlab-ctl stop sidekiq
docker compose exec gitlab gitlab-ctl status
```

Der Wiederherstellungsprozess kann dann folgendermaßen durchgeführt werden

```
docker compose exec gitlab gitlab-backup restore BACKUP=<BACKUP_NAME>
```

Wobei `<BACKUP_NAME>` dem Dateinamen des Archivs entspricht.

Ist der Name des Archivs `1678714058_2023_03_13_15.8.4_gitlab_backup.tar` würde der vollständige Befehl folgendermaßen aussehen:

```
docker compose exec gitlab gitlab-backup restore
BACKUP=1678714058_2023_03_13_15.8.4
```

Der Teil mit `_gitlab_backup.tar` muss dabei nicht angegeben werden, aber im Dateinamen enthalten sein!

Nach erfolgreicher Wiederherstellung kann einfach der Container durchgestartet werden.

```
docker compose restart gitlab
```

Zur Sicherheit noch einen abschließenden Check

```
docker compose exec gitlab gitlab-rake gitlab:check SANITIZE=true
```

Firewall und Fail2Ban

Da wir Docker verwenden, ist bereits `iptables` im Einsatz. Docker nutzt iptables unter anderem für die Port-Weiterleitungen zwischen Host und Container.

Eine etwas einfachere Syntax bietet unter Debian/Ubuntu der Wrapper ufw. Eine Alternative, speziell unter RedHat basierenden Distributionen ist firwalld.

```
apt install ufw
```

Bevor wir `ufw` aktivieren, sollten wir die gewünschten Ports (vor allem SSH für den weiteren Zugriff auf den Host) erlauben.

```
ufw allow ssh
ufw allow http
ufw allow https
ufw allow 2222/tcp
```

Danach können wir die Regeln aktivieren

```
ufw enable
```

Wir können jederzeit den Status abrufen

```
ufw status
```

`ufw` setzt einfach `iptables` Regeln, die wir Alternativ auch verifizieren können

```
iptables --list
```

Neben der Firewall bietet sich noch an das System mittels `fail2ban` abzusichern. `fail2ban` analysiert Logs und erstellt ggf. Firewall-Regeln, um potentielle Angreife aufzusperren.

```
apt install fail2ban
```

Die Konfigurationsdateien von `fail2ban` finden sich unter `/etc/fail2ban` (am Host!)

Um `fail2ban` zu aktivieren, müssen wir im ersten Schritt die vorhandene Default-Konfigurationsdatei `jail.conf` auf `jail.local` kopieren.

```
cd /etc/fail2ban
cp jail.conf jail.local
```

In `jail.local` können wir nun unsere angepassten Konfigurationen vornehmen.

Es empfiehlt sich eine Liste mit ausgenommenen IP-Adressen zu setzen, um sich nicht selbst ungewollt auszusperren. Fürs erste setzten wir einmal nur Localhost.

/etc/fail2ban/jail.local

```
ignoreip = 127.0.0.1/8 ::1
```

Unter der `banaction` können wir die Aktion festlegen, die fürs bannen genutzt wird. Per Default ist das `iptables-multiport`. Es gibt auch für `ufw` vordefinierte Aktionen. Diese nützen aber beim Einsatz von Docker nichts, da Docker in `iptables` selbst Regeln festlegt. Aus diesem Grund belassen wir hier die `iptables` Aktionen.

/etc/fail2ban/jail.local

```
banaction = iptables-multiport
banaction_allports = iptables-allports
```

Es findet sich hier auch bereits eine Konfiguration für GitLab. Diese müssen wir anpassen, bzw. werden wir entfernen/auskommentieren und eine separate Konfiguration festlegen.

/etc/fail2ban/jail.local

```
#[gitlab]
#port    = http,https
#logpath = /var/log/gitlab/gitlab-rails/application.log
```

Anstelle der nun auskommentierten Konfiguration in `jail.local` fügen wir einfach unter `/etc/fail2ban/jail.d/` eine Konfiguration für gitlab hinzu.

```
touch /etc/fail2ban/jail.d/gitlab.conf
```

/etc/fail2ban/jail.d/gitlab.conf

```
[gitlab]
enabled = true
port = http,https
filter = gitlab
chain = DOCKER-USER
logpath = /root/gitlab/logs/gitlab-rails/application.log
maxretry = 3
```

Für gitlab gibt es bereits einen vordefinierten Filter, den wir hier angeben. Wichtig ist die `chain = DOCKER-USER` zu setzten, damit die passenden Regeln in iptables gesetzt werden.

Wir können `fail2ban` nun aktivieren und testen.

```
systemctl restart fail2ban
```

Zum testen machen wir einfach mehrere fehlschlagende Loginversuche auf GitLab.

Wir können mit `fail2ban-client` den Status überprüfen.

```
$ fail2ban-client status gitlab
Status for the jail: gitlab
|- Filter
|  |- Currently failed: 1
|  |- Total failed:     6
|  `- File list:        /root/gitlab/logs/gitlab-rails/application.log
`- Actions
   |- Currently banned: 1
   |- Total banned:     2
   `- Banned IP list:   192.168.56.1
```

In iptables sollten wir eine entsprechende Regel finden.

```
$ iptables --list DOCKER-USER
Chain DOCKER-USER (1 references)
target     prot opt source               destination
f2b-gitlab  tcp  --  anywhere             anywhere             multiport
dports http,https
RETURN     all  --  anywhere             anywhere
$ iptables --list f2b-gitlab
Chain f2b-gitlab (1 references)
target     prot opt source               destination
REJECT     all  --  192.168.56.1         anywhere             reject-with
icmp-port-unreachable
RETURN     all  --  anywhere             anywhere
```

Um die IP wieder freizugeben, nutzen wir wieder den `fail2ban-client`

```
$ fail2ban-client set gitlab unbanip 192.168.56.1
1
```

Nun sollten wir wieder Zugriff bekommen.

> GitLab selbst checkt auch Fehlgeschlagene Logins und sperrt ggf. Benutzer. Damit ist zwar weiterhin ein Zugriff auf die Website möglich (zB für zusätzliche Loginversuche), aber der Benutzer selbst ist gesperrt und kann sich nicht mehr einloggen. Diese können als Administrator wieder freigegeben werden, bzw. sollte man sich als Admin aussperren geht das auch über die Commandline.
>
> - unlock user
> - reset user passwort

Weitere Punkte

HTTPS mit Let's Encrypt

GitLab kann sich selbst um die TLS Zertifikate für HTTPS kümmern, mithilfe von Let's Encrypt.

Alles was dazu notwendig ist, sind folgende Einstellungen.

docker-compose.yml

```
# ...
    environment:
      GITLAB_OMNIBUS_CONFIG: |
        external_url 'https://gitlab.dih.at'
        letsencrypt['enable'] = true
        letsencrypt['contact_emails'] = ['your.email@here']
        letsencrypt['auto_renew_hour'] = "12"
        letsencrypt['auto_renew_minute'] = "30"
        letsencrypt['auto_renew_day_of_month'] = "*/7"
# ...
```

Der Server muss dafür natürlich im WWW erreichbar sein.

Alternativ können auch eigene Zertifikate genutzt werden. Siehe dazu die Dokumentation

Docker Rootless

Docker als Systemdienst zu betreiben hat einen gravierenden Nachteil. Um die Container betreiben zu können ist der Root-User nötig. Entweder via `sudo` oder als Teil der `docker`-Gruppe.

Seit einiger Zeit bietet Docker auch eine Rootless Variante an. Dabei läuft der nötige Docker-Daemon als Dienst auf Benutzer-Ebene.

GitLab kann auch damit Problemlos betrieben werden, es sind aber einige Punkte dabei zu beachten:

- Binding auf Port 80, 443 etc. (alle Ports unter 1024) sind nicht möglich
- Direkter Zugriff auf die Dateien am Host-System ist ohne Root-User (`sudo`) nicht mehr oder nur sehr eingeschränkt möglich. Hier bleibt einzig die Möglichkeit mittels `docker exec` und `docker cp` mit dem Container Sinnvoll zu interagieren.

LDAP und andere Alternative Authentifizierungsmöglichkeiten

Sollte es schon einen bestehenden LDAP-Server für die Benutzer-Authentifizierung im Unternehmen geben, kann dieser einfach mit GitLab verbunden werden. Siehe dazu GitLab LDAP Integration.

Neben LDAP stehen noch weitere Dienste wie Auth0, Azure oder OmniAuth zur Verfügung. Für eine Vollständige Liste siehe hier.

GitLab hinter Reverse-Proxy

Verwendet man Alternative Ports für HTTP/HTTPS aber möchte trotzdem GitLab über die Standard-Ports nach aussen hin erreichbar haben, bietet sich ein vorgeschalteter Reverse-Proxy an.

Im folgenden eine Beispielkonfiguration für einen Apache Webserver, der öffentlich über HTTP und HTTPS erreichbar ist und die Anfragen als Reverse-Proxy an GitLab weiterleitet. GitLab läuft in diesem Beispiel am selben Host.

Apache Konfiguration:

```
<VirtualHost *:80>
  ServerAdmin  admin@dih.at
  ServerName   gitlab.dih.at
  DocumentRoot /var/www/gitlab/html

  UseCanonicalName on

  ErrorLog  /var/log/gitlab/apache_error.log
  CustomLog /var/log/gitlab/apache_access.log combined

  ProxyPass        / http://127.0.0.1:10080/ timeout=600
  ProxyPassReverse / http://127.0.0.1:10080/

  <Proxy *>
    Order deny,allow
    Allow from all
    ProxyPreserveHost on
  </Proxy>

  <IfModule mod_ssl.c>
    RewriteEngine on
    RewriteCond %{SERVER_NAME} =gitlab.dih.at
    RewriteRule ^ https://%{SERVER_NAME}%{REQUEST_URI} [END,NE,R=permanent]
  </IfModule>
</VirtualHost>
```

```
<IfModule mod_ssl.c>
<VirtualHost *:443>
  ServerAdmin  admin@dih.at
  ServerName   gitlab.dih.at
  DocumentRoot /var/www/gitlab/html

  UseCanonicalName on

  ErrorLog  /var/log/gitlab/apache_error.log
  CustomLog /var/log/gitlab/apache_access.log combined

  RequestHeader unset Accept-Encoding
  RequestHeader set Host "gitlab.dih.at"
  RequestHeader add X-Forwarded-Ssl on
  RequestHeader set X-Forwarded-Proto "https"

  ProxyPass        / http://127.0.0.1:10080/ nocanon timeout=600
  ProxyPassReverse / http://127.0.0.1:10080/

  AllowEncodedSlashes NoDecode

  <Proxy *>
    Order deny,allow
    Allow from all
    ProxyPreserveHost on
  </Proxy>

  SSLCertificateFile    /etc/apache2/ssl/gitlab.dih.at.crt
  SSLCertificateKeyFile /ect/apache2/ssl/gitlab.dih.at.key
</VirtualHost>
</IfModule>
```

Mittels `ProxyPass` und `ProxyPassReverse` werden die Anfragen an *gitlab.dih.at* an http://127.0.0.1:10080/ weitergeleitet. Die `RequestHeader` Einträge sind nötig, damit die passenden Header für GitLab weitergesendet werden.

GitLab selbst läuft in diesem Beispiel ausschließlich via HTTP, HTTPS ist deaktiviert. Die Konfiguration für GitLab ist folgendermaßen dabei angepasst.

docker-compose.yml

```
# ...
  environment:
    GITLAB_OMNIBUS_CONFIG: |
      external_url 'https://gitlab.dih.at'
      nginx['enable'] = true
      nginx['listen_port'] = 80
      nginx['listen_https'] = false
      nginx['redirect_http_to_https'] = false
  ports:
    - '10080:80'
# ...
```

HTTPS und vor allem der Redirect von HTTP auf HTTPS wird Konfigurations-Seitig deaktiviert, da dies nun durch den vorgeschalteten Reverse-Proxy passiert.