Server - Anleitung zum selber kompilieren

Setup auf Linux

> Diese Setup-Dokumentation ist am Beispiel eines Raspberry Pi 3 mit Raspberry Pi OS (64 Bit) erstellt, eignet sich in gleicher Weise jedoch auch für Debian, Ubuntu, etc.

Wir flashen auf einem beliebigen Rechner eine neue SD-Karte mit Hilfe von `rpi-imager`.

Siehe https://www.raspberrypi.com/software.

> Ein fertig eingerichtetes Image kann unter folgender Seite gefunden werden: Server - Raspberry Pi Image flashen / FabAccess Wander-Setup.

Unser Raspberry Pi 3 Model B+ ist vergleichsweise ziemlich alt und hat folgende Specs:

Raspberry Pi 3 B+ Specs

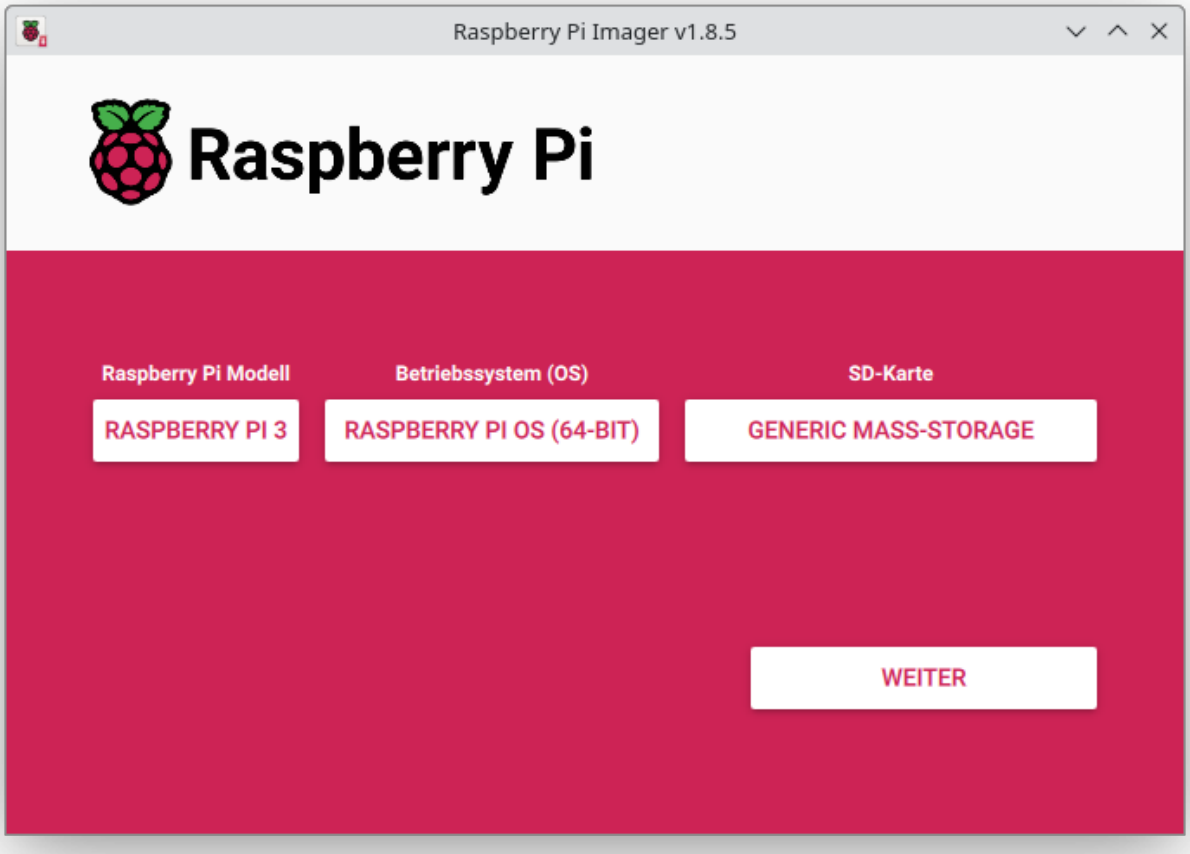
- Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-Bit-SoC @ 1,4GHz
- 1GB LPDDR2 SDRAM
- 2,4GHz und 5GHz IEEE 802.11.b/g/n/ac Wireless LAN, Bluetooth 4.2, BLE
- Gigabit Ethernet über USB 2.0 (maximaler Durchsatz 300 Mbps)
- Erweiterte 40-polige GPIO-Stiftleiste
- HDMI® in voller Größe
- 4 USB 2.0-Anschlüsse
- CSI-Kamera-Port für den Anschluss einer Raspberry Pi-Kamera
- DSI-Display-Port für den Anschluss eines Raspberry Pi-Touchscreen-Displays
- 4-poliger Stereoausgang und Composite-Video-Anschluss
- Micro-SD-Anschluss zum Laden Ihres Betriebssystems und Speichern von Daten
- 5V/2.5A DC Stromeingang
- Power-over-Ethernet (PoE)-Unterstützung (separater PoE-HAT erforderlich)

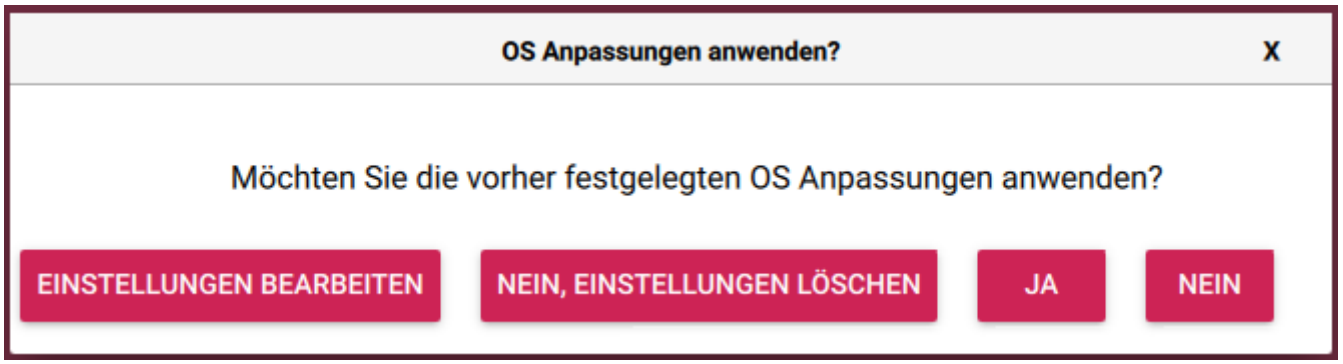
Raspberry Pi Imaging / Provisioning

Wir installieren den Imager, um damit das Raspberry Pi OS aufzuspielen:

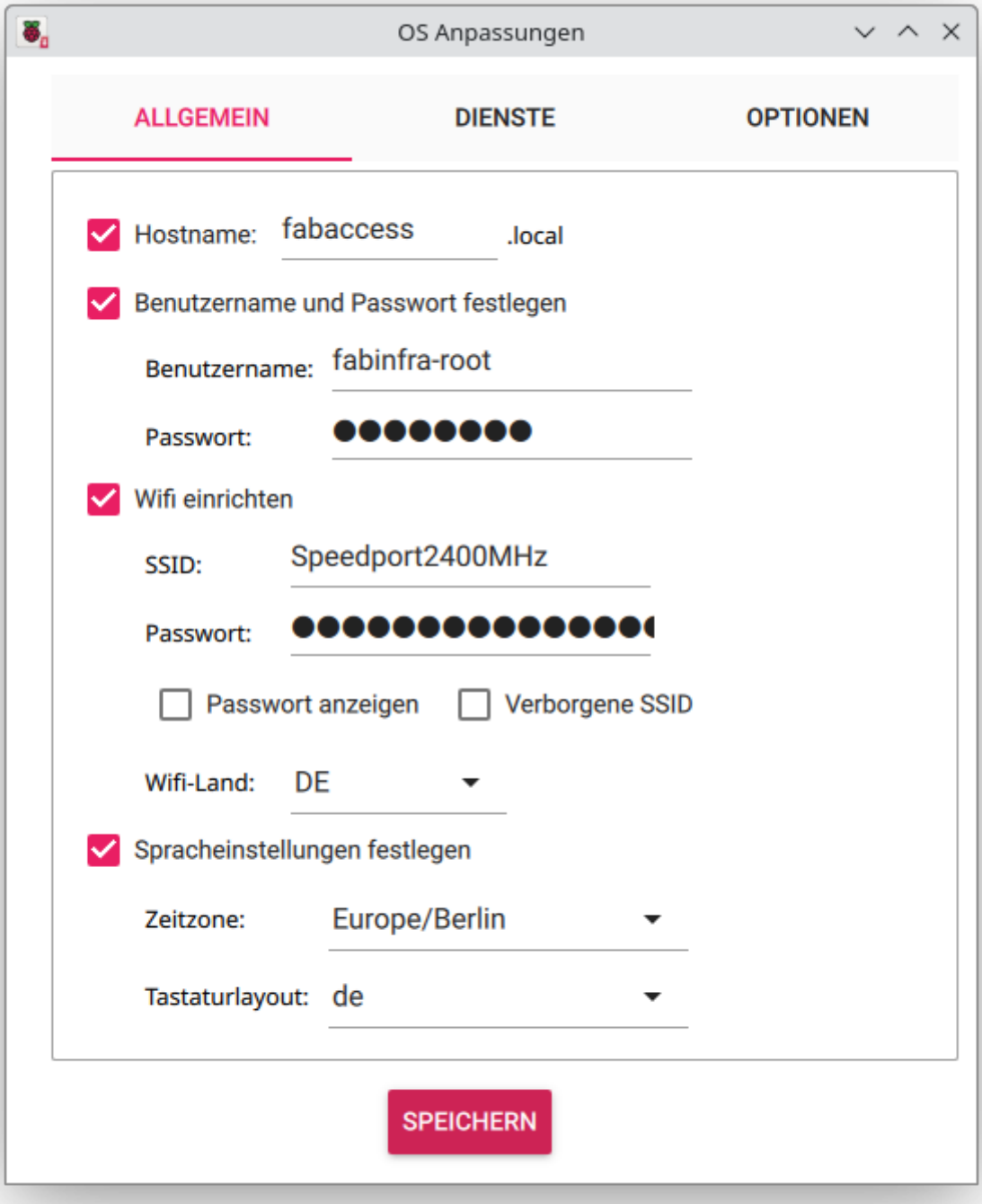
```
sudo apt install rpi-imager
```

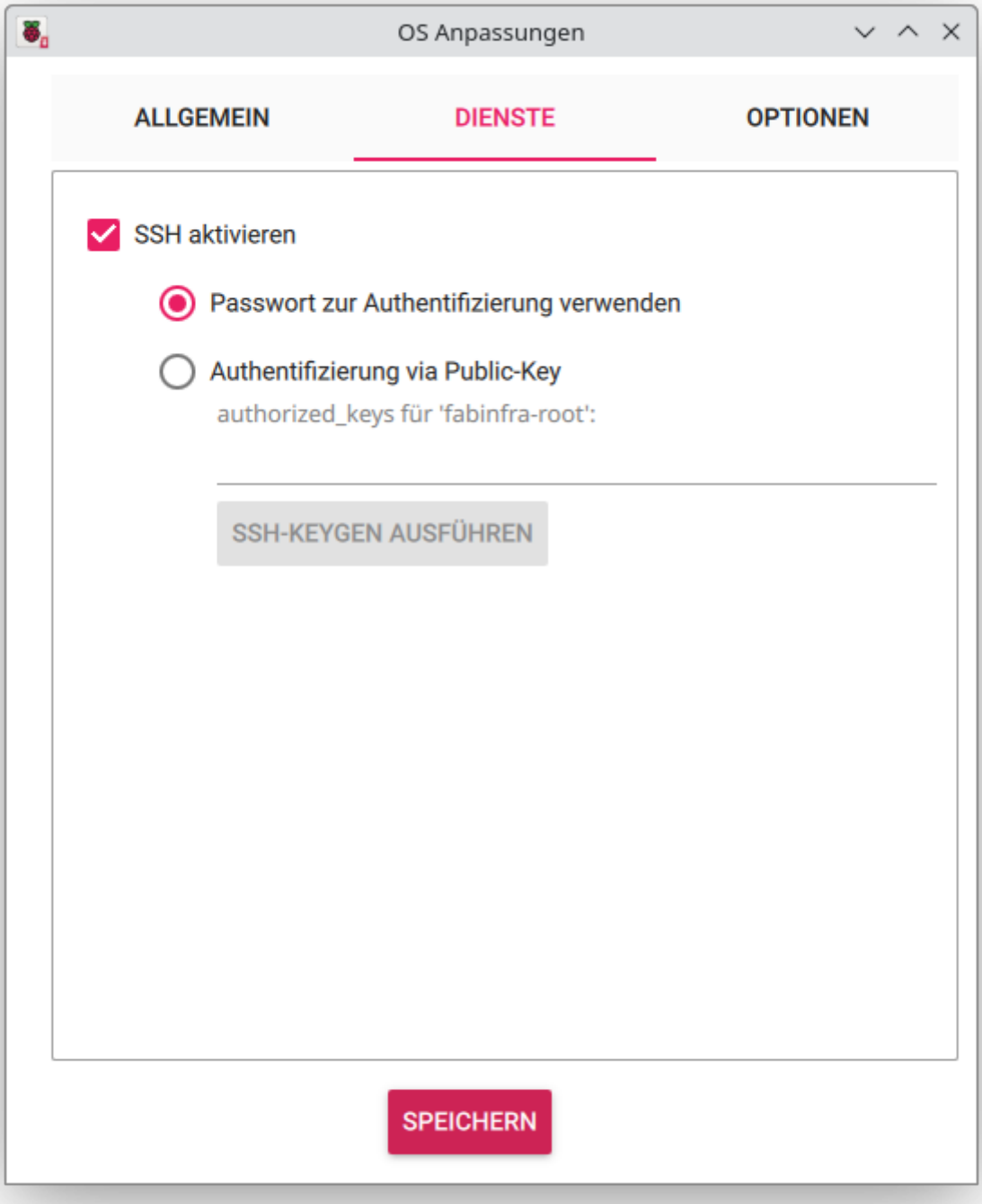
Setup-Schritte mit Screenshots zeigen (aufklappen)

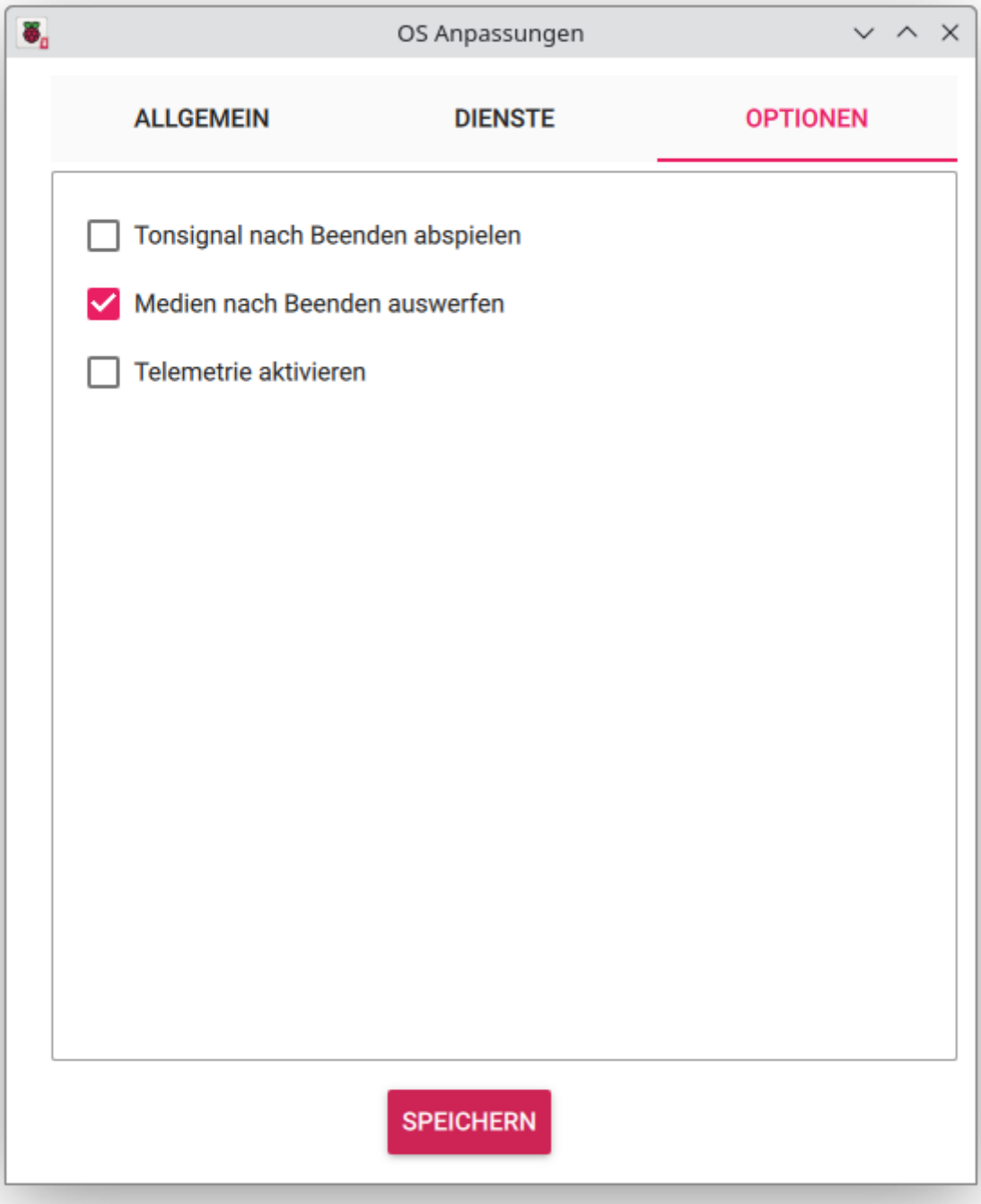




Wir vergeben als Benutzername `fabinfra-root` und das Passwort `vulca`


OS Anpassungen
ALLGEMEIN
DIENSTE
OPTIONEN
Hostname:
fabaccess
.local
Benutzername und Passwort festlegen
Benutzername:
fabinfra-root
Passwort:
Wifi einrichten
SSID:
Speedport2400MHz
Passwort:
Passwort anzeigen
Verborgene SSID
Wifi-Land:
DE
Spracheinstellungen festlegen
Zeitzone:
Europe/Berlin
Tastaturlayout:
de
SPEICHERN


OS Anpassungen
ALLGEMEIN
DIENSTE
OPTIONEN
SSH aktivieren
Passwort zur Authentifizierung verwenden
Authentifizierung via Public-Key
authorized_keys für 'fabinfra-root':
SSH-KEYGEN AUSFÜHREN
SPEICHERN



Unser Image: Debian Bookworm Linux `MP PREEMPT Debian 1:6.6.51-1+rpt3 (2024-10-08) aarch64`

Standardpakete installieren

Wir installieren zunächst Pflichtbibliotheken. Einige der folgenden Pakete sind unter Umständen bereits standardmäßig vorinstalliert. Wir geben sie der Vollständigkeit halber

trotzdem an.

```
sudo apt update && sudo apt upgrade
sudo apt install -y gsasl libgsasl7-dev libssl-dev libclang-dev build-essential cmake clang
capnproto mosquitto mosquitto-clients
```

Rust + Cargo installieren

Wir installieren rustc und cargo. Für die aktuellen Versionen siehe.

- https://releases.rs
- https://doc.rust-lang.org/nightly/cargo/CHANGELOG.html

> Zum Zeitpunkt der letzten Aktualierung dieser Doku wurde Rust 1.84.1 verwendet.

Systemeigenes Rust entfernen (optionaler Schritt)

Wir empfehlen die Verwendung von Rustup zum Installieren einer Rust-Umgebung (siehe nachfolgender Schritt).

```
# Zeige die aktuell installierten Versionen von rustc und cargo
rustc --version
cargo --version

rustc 1.63.0
cargo 1.65.0

# Zunächst entfernen wir das von Raspberry Pi mitgelieferte Rust
sudo apt purge rustc cargo libstd-rust-* libstd-rust-dev
```

Rust und Cargo per Rustup installieren

```
# Wir installieren nun das aktuelle Rust per rustup (als normaler Nutzer). Rustup erlaubt das
flexible Installieren beliebiger Rust-Versionen - dauert ca. 10 Minuten - wir installieren die
Standardvariante 1)
curl https://sh.rustup.rs -sSf | sh

# cargo in .bashrc einfügen und Umgebung neu laden
echo 'source "$HOME/.cargo/env"' >> ~/.bashrc
source ~/.bashrc
```

```
# wir prüfen, ob wir die aktuelle Rust Version haben
rustup show

#  oder installieren sie ...
rustup install stable
rustup default stable

# Update auf aktuelles Release. Falls die letzte Kompilation länger her ist
rustup update

# (optional) eine spezifische Version kann wie folgt installiert werden
rustup install 1.84.1
rustup default 1.84.1

# Version erneut checken
rustc --version
cargo --version

cargo 1.84.1 (66221abde 2024-11-19)
rustc 1.84.1 (e71f9a9a9 2025-01-27)

# Zum Herausfinden, welche genaue Version sich hinter "stable" befindet:
rustc +stable --version
cargo +stable --version
```

> Compile schlägt fehl? Sind alle Bibliotheken installiert? Wurde das Git-Repository rekursiv ausgecheckt (Abhängigkeiten)?

Mosquitto vorbereiten

```
sudo vim /etc/mosquitto/mosquitto.conf
```

Wir lassen Mosquitto auf IPv4 und IPv6 lauschen und binden es an alle Interfaces (`0.0.0.0` bzw `::`), da unsere MQTT Aktoren im Netzwerk sonst den Server nicht erreichen. Aus Sicherheitsgründen erlauben wir keine anonymen Zugriffe (`allow_anonymous false`). Außerdem legen wir eine Passwortdatei an, die mindestens einen Nutzer + Passwort enthält, gegen den sich sowohl der BFFH Server, als auch die MQTT Geräte später authentifizieren.

```
# Place your local configuration in /etc/mosquitto/conf.d/
# /usr/share/doc/mosquitto/examples/mosquitto.conf.example

# note: :: does not automatically configure 0.0.0.0
listener 1883 ::
listener 1883 0.0.0.0

allow_anonymous false
password_file /etc/mosquitto/pw.file
pid_file /run/mosquitto/mosquitto.pid
persistence true
persistence_location /var/lib/mosquitto/
#log_dest file /var/log/mosquitto/mosquitto.log
#log to journald
log_dest syslog
include_dir /etc/mosquitto/conf.d
```

Passwortdatei anlegen

Wir erzeugen einen Standardnutzer und nennen ihn `fabaccess-defaultuser` (kann beliebig genannt werden) und vergeben ein Passwort. Wir verwenden als Passwort den Wert `default`.

```
sudo mosquitto_passwd -c /etc/mosquitto/pw.file fabaccess-defaultuser
sudo chmod 400 /etc/mosquitto/pw.file
```

> Es können beliebig viele Nutzer in die Passwortdatei geschrieben werden. Das hängt vom gewünschten Setup ab!

Berechtigungen anpassen

```
sudo chown -R mosquitto:mosquitto /etc/mosquitto/
```

Mosquitto Dienst starten

```
sudo systemctl restart mosquitto.service
sudo journalctl -f -u mosquitto.service
```

Je nach Konfiguration von Mosquitto finden wir die Log-Einträge nun entweder per journalctl im syslog oder in einer separaten Datei.

Swap File erhöhen

BFFH benötigt zum Kompilieren mehr als 1 GB RAM. Ein Raspberry Pi 3 hat nur 1 GB. Dadurch stürzt er beim Kompilieren mit OOM-Fehlern ab. Deshalb erhöhen wir den Swap temporär auf 2 GB. Dieser Trick basiert auf

https://www.alibabacloud.com/blog/optimization-on-memory-usage-during-rust-cargo-code-compiling_601189

```
sudo su
dphys-swapfile swapoff
echo "CONF_SWAPSIZE=2048" > /etc/dphys-swapfile
dphys-swapfile setup
dphys-swapfile swapon

# Überprüfen von "Swp" mit htop
htop
```

BFFH Nutzer anlegen

Wir legen einen gesonderten Nutzer zum Ausführen von BFFH an. Es ist ein Systembenutzer ohne eigenes Homeverzeichnis (`-s` Parameter).

```
sudo useradd -m -s /bin/bash bffh
```

BFFH Git Projekt auschecken

> Zum Zeitpunkt der letzten Aktualierung dieser Doku wurde das **Release 0.4.3** genutzt.

Nach dem Installieren von Mosquitto können wir uns unserem BFFH Server widmen. Wir arbeiten nachfolgend im Verzeichnis `/opt/fabinfra/`. Es kann aber jedes andere Verzeichnis verwendet werden. Zunächst checken wir unser Git Projekt aus (und zwar **rekursiv**, da es Abhängigkeiten gibt!).

```
TGT="/opt/fabinfra"
mkdir -p $TGT/
```

```
cd  $TGT/
git clone https://gitlab.com/fabinfra/fabaccess/bffh.git --recursive
cd bffh/

# wir wechseln den Branch auf die aktuellste Entwicklerversion. Siehe
https://gitlab.com/fabinfra/fabaccess/bffh/-/commits/feature%2Fclaims_api -> das lassen wir,
denn es klappt nicht mit "cargo build --release"
#git checkout feature/claims_api

# wir wechseln zum Branch "development"
git checkout development

# Submodules updaten
git submodule update --remote
```

BFFH testen / entwickeln

Wir müssen BFFH zum Code-Testing nicht als optimierte Binary kompilieren. Das Ausführen ist möglich über cargo und dafür stehen verschiedene Befehle zur Verfügung.

BFFH Hilfe anzeigen:

```
cargo run -- --help
```

`bffhd` mit Konfigurationsdatei starten

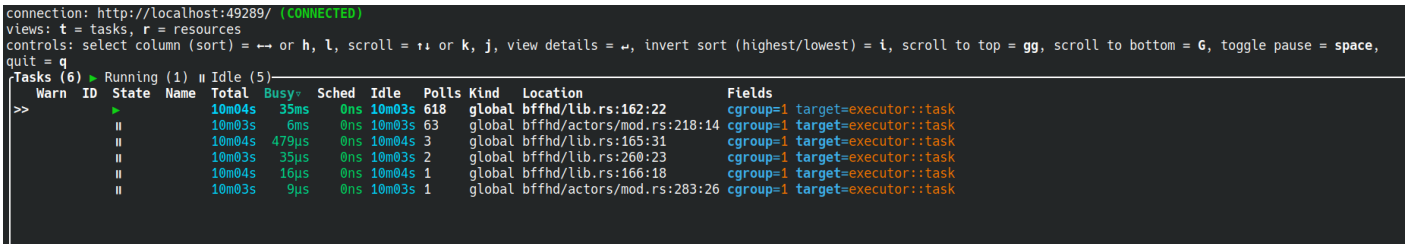
```
cargo run -- -c /etc/bffh/bffh.dhall
```

BFFH nutzt die Tokio Bibliothek und kann auch per Console inspiziert werden. Dafür kann über cargo der Befehl `tokio-console` installiert werden. Dieser Befehl wird automatisch in `$HOME/.cargo/bin/` installiert und steht dem Nutzer fortan global zur Verfügung.

```
cargo install --locked tokio-console
```

Bei laufendem bffhd Prozess können wir uns dann einklinken:

```
tokio-console http://localhost:49289
```



> Weitere Infos zu Logging und Debugging

BFFH kompilieren

BFFH mit `release`-Flag zu einer ausführbaren Datei kompilieren, um optimierte Compile-Artefakte zu erzeugen - auf einem leistungsstarken PC dauert das ca. 2 Minuten, auf einem Raspberry Pi 3 zwischen 30 Minuten und 2,5 Stunden. Die fertige Binary liegt danach in `target/release/bffhd`.

```
cargo build --release
```

Wir können auch einen erweiteren Build erzeugen, der debuginfo erhält. Die erzeugte Binary findet sich dann in `target/debug/bffhd`.

```
RUSTFLAGS=-g cargo build --release
#    Finished release [optimized + debuginfo] target(s) in 2m 42s
```

> Die erzeugte Binary `bffhd` ist ca. 200 MB groß! Es ist möglich diese Binary deutlich zu verkleinern, jedoch werden in der Regel hilfreiche Debug Symnole entfernt.

BFFH installieren

Die fertige Binary kopieren ein Linux-typisches Standardverzeichnis (`/usr/bin`) kopieren und Rechte anpassen:

```
# Installiere alle kompilierten Binaries in das Zielverzeichnis
cargo install --path /usr/bin/
```

oder per gewöhnlichem Kopierbefehl:

```
cp /opt/fabinfra/bffh/target/release/bffhd /usr/bin
sudo chown root:root /usr/bin/bffhd
```

BFFH konfigurieren

Die Konfiguration von BFFH wird in diesem Kapitel vorbereitet und grundlegend erzeugt. Die Feinjustageschritte sind im dedizierten Kapitel FabAccess Konfiguration beschrieben.

Verzeichnisse anlegen und Besitzer anpassen

```
sudo mkdir -p /etc/bffh/
sudo mkdir -p /etc/bffh/certs/
sudo mkdir -p /var/lib/bffh/ #hier wird die Datenbank gespeichert

sudo chown bffh:bffh /etc/bffh/
sudo chown bffh:bffh /etc/bffh/certs/
sudo chown bffh:bffh /var/lib/bffh/

sudo chmod 750 /etc/bffh/
sudo chmod 750 /etc/bffh/certs/
sudo chmod 750 /var/lib/bffh/
```

Zertifikat erzeugen (self signed)

Das Zertifikat benötigen wir für den BFFH Server, da wir es in der Konfigurationsdatei angeben müssen. Es muss vom Typ X.509 Version 3 sein.

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/bffh/certs/bffh.key -out
/etc/bffh/certs/bffh.crt

# Wir füllen die Fragen mit Beispielwerten aus:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:DE
State or Province Name (full name) [Some-State]:Saxony
Locality Name (eg, city) []:FabInfra Headquarter
Organization Name (eg, company) [Internet Widgits Pty Ltd]:FabInfra Community
Organizational Unit Name (eg, section) []:Demo
Common Name (e.g. server FQDN or YOUR name) []:fabaccess.local
```

```
Email Address []:webmaster@fab-access.org
```

Wir prüfen die geforderte Version **X.509** v3 des Zertifikats:

```
sudo openssl x509 -in /etc/bffh/certs/bffh.crt -noout -text | grep "Version"
```

Das Ergebnis muss lauten: `Version: 3 (0x2)`

Üblicherweise sollten Zertifikate und Schlüssel nur von previligierten Benutzern gelesen werden können, deshalb passen wir Berechtigungen und Eigentümer an:

```
chown bffh:bffh /etc/bffh/certs/bffh.key
chown bffh:bffh /etc/bffh/certs/bffh.crt
chmod 640 /etc/bffh/certs/bffh.key
chmod 640 /etc/bffh/certs/bffh.crt
```

Konfiguration (bffh.dhall) anlegen

Wir legen ein neues Verzeichnis an und kopieren die Beispielkonfiguration dort hin.

```
cp /opt/fabinfra/bffh/examples/bffh.dhall /etc/bffh/bffh.dhall
```

```
vim /etc/bffh/bffh.dhall
```

Wir passen die Konfiguration nun an unsere Wünsche an. Die Referenz für die Konfiguration findest du hier.

Nutzerdatenbank (users.toml) anlegen und importieren

```
cp /opt/fabinfra/bffh/examples/users.toml /etc/bffh/users.toml
```

```
vim /etc/bffh/users.toml
```

Hier geht's zur Dokumentationen der users.toml

Nach dem Anlegen der Nutzerdatenbank importieren wir diese in BFFH mit folgendem Kommando (dabei ist auch die Hauptkonfiguration `bffh.dhall` anzugeben!). Alternativ kann auch ein Script verwendet werden, welches ein paar Pre-Checks ausführt.

```
/usr/bin/bffhd --config /etc/bffh/bffh.dhall --load /etc/bffh/users.toml
```

> Bei jeder Änderung der `users.toml` Datei muss ein Import durchgeführt werden, da BFFH die Änderungen nicht automatisch einliest!

Berechtigungen und Besitzer anpassen

Damit nur Nutzer und Gruppe, nicht aber Externe auf die Dateien zugreifen können, beschränken wir den Zugriff auf unseren Ordner `/opt/fabinfra`. In diesem Verzeichnis befinden sich nur Dienste und Configs, die von BFFH ausgeführt werden sollen. Sollten Dateien nachträglich hinzugefügt oder modifiziert werden ist darauf zu achten, dass der Eigentümer und die Berechtigungen entsprechend angepasst werden.

```
BFFH_DIR="/opt/fabinfra/"
sudo find $BFFH_DIR -type d -exec chmod 750 {} +
sudo find $BFFH_DIR -type f -exec chmod 640 {} +
chown -R bffh:bffh $BFFH_DIR
```

Server starten (manueller Test)

Wir starten den Dienst `bffhd` manuell und dann prüfen wir, ob er auf dem richtigen Port und Interface lauscht:

```
/usr/bin/bffhd --verbose --config /etc/bffh/bffh.dhall --log-format Pretty


netstat -an | grep "LISTEN" | grep -v "LISTENING"
```

> Hinweis: `bffhd` erzeugt oder verwendet beim Starten zwei Dateien, und zwar eine Lock-Datei namens `bffh-lock` und eine interne Datenbankdatei `bffh`, die die Machine States und die Benutzer enthält, die aus `users.toml` stammen.

> BFFH startet standardmäßig auf dem TCP-Port `59661`, wenn nicht geändert.
>
> Außerdem wird im Hintergrund auf dem TCP-Port `49289` eine Debugger Console gestartet.

systemd Service anlegen und BFFH automatisch starten

Wenn unser manueller Test geklappt hat, dann sollten wir bffh als Dienst installieren und uns die Arbeit abnehmen lassen:

```
vim /etc/systemd/system/bffh.service
```

```
[Unit]
Description=FabAccess BFFH Service
After=network.target

[Service]
Type=simple
User=bffh
Group=bffh
ExecStartPre=/usr/bin/bffhd --check --config /etc/bffh/bffh.dhall
Environment="BFFH_LOG=warn"
ExecStart=/usr/bin/bffhd --verbose --config /etc/bffh/bffh.dhall --log-format Pretty
Restart=on-failure
RestartSec=30
LogsDirectoryMode=750
LogsDirectory=bffh

[Install]
WantedBy=multi-user.target
```

Dienst aktivieren, starten und prüfen

```
sudo systemctl daemon-reload
sudo systemctl enable bffh.service --now
journalctl -f -u bffh.service
```

> **Achtung beim Naming:** `bffhd` ist unser BFFH **Daemon** und impliziert mit seinem Namen, dass das Programm permanent laufen soll. Wir verwenden nachfolgend systemd zum Installieren von `bffhd` als Service. Den Service nennen wir jedoch `bffh.service` - so, wie der Serverdienst heißt.

Wir erlauben unserem Nutzer bffh außerdem den Dienst selbstständig zu starten und zu stoppen:

```
sudo echo "bffh ALL=NOPASSWD: /usr/bin/systemctl start bffh.service" > /etc/sudoers.d/bffh
sudo echo "bffh ALL=NOPASSWD: /usr/bin/systemctl stop bffh.service" > /etc/sudoers.d/bffh
sudo echo "bffh ALL=NOPASSWD: /usr/bin/systemctl restart bffh.service" > /etc/sudoers.d/bffh
```

Wenn wir uns als bffh Nutzer einloggen, können wir dann ausführen:

```
sudo su - bffh


# als bffh Nutzer
sudo systemctl start bffh.service
sudo systemctl stop bffh.service
sudo systemctl restart bffh.service
```

Nach dem Starten von BFFH haben wir im Idealfall einen stabilen Dienst am Laufen, welcher möglichst keine Fehlerausgaben produziert. Falls doch, sollten wir in die Logs schauen bzw. vorher prüfen, wie die Log-Ausgabe konfiguriert ist.

> **Log-Größe von journalctl beachten:** Je nach Konfiguration des Log Levels von BFFH und der Anzahl der aktiven Nutzer kann die Logdatei innerhalb von 1-2 Wochen auf über 5 GB und mehr anwachsen und insbesondere bei Systemen auf Basis von SD-Karten zum Defekt des Schreibmediums führen, denn diese altern durch die vielen Schreibzyklen sehr schnell. im produktiven Fall ist deshalb zum einen darauf zu achten, dass die SD-Karte stets überlebt (Log Level auf `Info` setzen), und dass das System nicht voll läuft. Hier ein paar Kommandos, um das Journal zu leeren (dies schützt nicht vor den Schreibzyklen, aber erlaubt eine bessere Übersicht).

```
sudo journalctl --rotate
sudo journalctl --vacuum-time=1s
```

Auch syslog läuft unter Umständen voll und kann geleert werden. Das ist allerdings ein sehr unschöner Trick und sollte nur zu absoluten Testzwecken angewendet werden:

```
sudo truncate -s 0 /var/log/syslog
```

Arch Linux

```
sudo pacman -Syu
sudo pacman -S make cmake clang gsasl
sudo pacman -S git rust capnproto
```

> Todo ...

CentOS

```
sudo yum update
sudo yum install curl && curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
sudo yum install epel-release && sudo yum install capnproto
sudo yum install https://packages.endpointdev.com/rhel/7/os/x86_64/endpoint-repo.x86_64.rpm &&
sudo yum install git
sudo yum install centos-release-scl && yum install llvm-toolset-7 && scl enable llvm-toolset-7
bash (Change bash to youre shell)
sudo yum install gcc-c++ libgsasl-devel openssl-devel cmake
```

> Todo ...

Version #98
Erstellt: 23 Oktober 2024 22:56:38 von Mario Voigt (Stadtfabrikanten e.V.)
Zuletzt aktualisiert: 22 März 2025 22:06:44 von Mario Voigt (Stadtfabrikanten e.V.)