

Hauptkonfiguration - bffh.dhall

BFFH verwendet Dhall für die Struktur von Ressourcen (z.B. Maschinen), Rollen, Akteuren und Initiatoren. Die Konfiguration von BFFH befindet sich in der Datei `bffh.dhall`. Die Datei kann auch umbenannt werden. Wichtig ist, dass sie dann überall korrekt referenziert wird (z.B. in Service Scripts). Der betreffende Quellcode findet sich in GitLab.



Offizielles Dhall Logo

Tipps & Tricks in Dhall

Dhall ist eine umfangreiche Sprache für die Konfiguration und erlaubt äußerst komplexe Datenstrukturen. Vieles davon wird für FabAccess vermutlich nur im kleineren Rahmen benötigt, jedoch besteht die Kompatibilität auch für komplizierte Gebilde. Im Rahmen dieser Dokumentationen gehen wir nur auf grundlegende, wichtige und besonders praktische Konfigurationsaspekte ein.

Ein nutzbringenden Cheatsheet für Dhall findet sich unter <https://docs.dhall-lang.org/howtos/Cheatsheet.html>

Literal für leere Einträge

Dhall-Records können auch leer übergeben werden. Dafür gibt es eine spezielle Syntax:

`{=}`. Siehe <https://hackage.haskell.org/package/dhall-1.42.1/docs/Dhall-Tutorial.html#g:6>

Beispiel:

```
initiators = {=}
```

Literal "Some"

Die folgenden beiden Angaben sind inhaltlich gleich. Das Wörtchen "Some" macht in unserem Beispiel den Wert `port` optional. Diese Notation kann verwendet werden, um innerhalb einer Konfiguration darauf hinzuweisen, dass ein Wert fest gesetzt werden könnte, aber aktuell nicht von Nöten ist. Siehe <https://hackage.haskell.org/package/dhall-1.42.1/docs/Dhall-Tutorial.html#g:5>

Beispiel:

```
listens = [{ address = "127.0.0.1" }]
```

```
listens = [{ address = "127.0.0.1", port = Some 59661 }]
```

Kommentare

Siehe <https://docs.dhall-lang.org/tutorials/Language-Tour.html#comments>

Zeilen, die mit `--` beginnen, werden automatisch als Kommentar gewertet. Das kann auch verwendet werden, um z.B. nicht mit dem Literal `"Some"` zu arbeiten. So können wir z.B. eine Originalzeile auskommentieren und darunter unsere angepasste Zeile schreiben.

Beispiel Zeilenkommentar:

```
--listens = [{ address = "127.0.0.1", port = 59661 }]  
listens = [{ address = "127.0.0.1" }]
```

Kommentare können auch in Blöcken geschrieben werden, wenn sie in einem Block beginnend `{-` und endend mit `-}` stehen. So werden diese bei etwaigen Formatieroperationen mit `dhall format <Datei>` nicht entfernt!

Beispiel Blockkommentar:

```
{- Main configuration file for bffh  
=====
```

In this configuration file you configure almost all parts of how bffh operates, but most importantly:

- * Machines
- * Initiators and Actors
- * Which Initiators and Actors relate to which machine(s)
- * Roles and the permissions granted by them

```
-}
```

Beispiel Blockkommentar an beliebiger Stelle:

```
roles = ./roles.dhall, {- Kommentar mitten im Text -}
```

Zeichenketten erzeugen

Werte können zusammengefügt werden (concatenation). Das geht zum Beispiel mit dem Syntax `++` wie folgt:

```
certfile = env:BFFH_CFG_PATH as Text ++ "/cert/bffh.crt"
```

Oder als Verkettungsbefehl:

```
hobbies = concatSep " ", " [ "piano", "reading", "skiing" ]
```

Dhall-Konfigurationen verschlanken

Folgende Möglichkeiten der Vereinfachung und Granulierung von Dhall-Dateien stellen wir vor. Wir berufen uns unter anderem auf die Dokumentation <https://github.com/dhall-lang/dhall-lang/blob/master/standard/imports.md>

Variablen/Konstanten innerhalb einer Datei setzen

Folgendes Beispiel erzeugt ein Listing `VARS`, welches drei Variablen enthält, die vereinheitlicht angesprochen werden können. Variablen lassen sich auf diese Art und Weise beliebig tief schachteln. Zu beachten ist das kleine Wörtchen `in`, welches im darauffolgenden Block (Record) eine Ersetzung bzw. Referenzierung anstrebt.

```
let VARS = {  
  BFFH_CFG_PATH = "/etc/bffh/",  
  MQTT_USER = "fabinfra101",  
  MQTT_PASSWD = "fablocal"  
}  
in {  
  listens = [  
    { address = "0.0.0.0", port = 59661 }  
  ],  
  certfile = VARS.BFFH_CFG_PATH ++ "certs/bffh.crt",
```

```
keyfile = VARS.BFFH_CFG_PATH ++ "certs/bffh.key",
mqtt_url = "mqtt://" ++ VARS.MQTT_USER ++ ":" ++ VARS.MQTT_PASSWD ++ "@0.0.0.0:1883",
db_path = VARS.BFFH_CFG_PATH ++ "bffh.db"
}
```

Umgebungsvariablen verwenden

Wir setzen z.B. in einer Bash-Umgebung einen Variablenwert und führen ein Kommando aus, welches die nachfolgende Dhall-Datei verwendet. Der Prozess, der die Dhall-Datei interpretiert, sucht in seiner übergebenen Umgebung (Environment) nach passenden Variablen und ersetzt die Werte entsprechend.

Als Rohtext in Dhall:

```
# bash
export BFFH_CFG_PATH="/etc/bffh"
```

```
bffh_cfg = env:BFFH_CFG_PATH as Text
```

oder als aufgelöster Pfad in Dhall:

```
# bash
export BFFH_CFG_PATH="~/bffh"
```

```
bffh_cfg = env:BFFH_CFG_PATH as Location
```

Inhalte aus URLs importieren

Folgender Syntax lädt den Content der entfernten URL als Variableninhalt:

```
let concatSep = http://prelude.dhall-lang.org/Prelude/Text/concatSep
sha256:fa909c0b2fd4f9edb46df7ff72ae105ad0bd0ae00baa7fe53b0e43863f9bd34a
```

Importe können mit Integritätsprüfungen geschützt werden, wenn ein SHA-256-Hash angehängt wird (wie beim obigen `concatSep`-Import). Dadurch verhindern wir Manipulation von außerhalb.

Dhall-Dateien innerhalb anderer Dhall-Dateien referenzieren (verschachteln)

Importierte Ausdrücke können transitiv andere Ausdrücke importieren. Eine exemplarische Datei `./schema.dhall` kann andere Dateien importieren:

```
{
  name : ./schema/name.dhall,
  age  : ./schema/age.dhall,
  hobbies : ./schema/hobbies.dhall
}
```

... und wenn `./schema/hobbies.dhall` einen relativen Import enthielte wie z.B.:

```
List ./hobby.dhall
```

... dann würde sich der relative Import von `./hobby.dhall` tatsächlich auf `./schema/hobby.dhall` beziehen. Dies ist als "Importverkettung" bekannt: die Auflösung von Importen relativ zur Position des aktuellen Ausdrucks.

Auf Kommas achten

In verschiedenen Konfigurationssprachen wie JSON, TOML und Dhall verhalten sich die Syntaxregeln unterschiedlich. In Dhall-Dateien ist es für einige Parser schädlich, wenn innerhalb einer List oder eines Records ein "trailing" Komma auftaucht. Folgendes sollte vermieden werden:

```
listens = [
  { address = "127.0.0.1" },
  { address = "::1" },
]
```

Syntaktisch korrekt ist:

```
listens = [
  { address = "127.0.0.1" },
  { address = "::1" }
]
```

BFFH kommt mit diesem Kommaproblem aus, jedoch der offizielle [dhall-haskell](#) Parser nicht.

Kurze oder lange Schreibweise

Wir können die Unterlemente `Dummy_123`, `Dummy_234` und `Dummy_345` im Beispiel in ein neues Klammernpaar schreiben oder direkt mit einem Punkt abtrennen und auf die gleiche Zeile schreiben:

```
actors = {  
  Dummy_123 = {  
    module = "Dummy",  
    params = {=}  
  },  
  Dummy_234 = {  
    module = "Dummy",  
    params = {=}  
  },  
  Dummy_345 = {  
    module = "Dummy",  
    params = {=}  
  }  
}
```

Oder entsprechend eingekürzt:

```
actors.Dummy_123 = {  
  module = "Dummy",  
  params = {=}  
},  
actors.Dummy_234 = {  
  module = "Dummy",  
  params = {=}  
},  
actors.Dummy_345 = {  
  module = "Dummy",  
  params = {=}  
}
```

Oder bei Rollen:

```
roles = {  
  Admin = {
```

```
permissions = [  
    "bffh.users.manage",  
    "bffh.users.info",  
    "bffh.users.admin"  
]  
}  
}
```

... eingekürzt:

```
roles.Admin.permissions = [  
    "bffh.users.manage",  
    "bffh.users.info",  
    "bffh.users.admin"  
]
```

Das eignet sich bei Rollen, Aktoren und Initiatoren! Wie es verwendet werden soll ist hier Geschmackssache.

Dhall-Dateien parsen, prüfen und formatieren

Dhall-Dateien können mit einer eigenen Bibliothek geparsed und auf Fehler geprüft werden. Das Resultat kann zudem neu formatiert ausgegeben werden - auch unter den Stichworten *Beautify* oder *Prettyprint* bekannt. Hierfür muss die Bibliothek `dhall` installiert werden.

```
sudo apt install dhall
```

Falls die Installation scheitert, kann das Paket auch manuell installiert werden (siehe <https://pkgs.org/search/?q=dhall>) - Beispiel für Raspberry Pi mit Debian Bookworm:

```
wget http://ports.ubuntu.com/pool/universe/h/haskell-dhall/dhall_1.32.0-1build1.1_arm64.deb  
dpkg -i dhall_1.32.0-1build1.1_arm64.deb
```

Das Folgende Kommando setzt zunächst die Konfigurationsdatei als Variable. Wenn diese Datei erfolgreich geparsed werden konnte, dann formatieren wir diese und überschreiben dann die Originaldatei - beim Scheitern unternehmen wir nichts.

Warnung:

Durch das automatische Formatieren werden alle etwaigen Kommentare entfernt (bekanntes Problem von Dhall, siehe [hier](#)), es sei denn diese werden in als Blockkommentar an den **Anfang** der Datei esetzt (siehe oben). Falls der [FabAccess Config Generator](#) genutzt wird, gehen dabei u.a. auch die Anfangs- und End-Tags `---` `||| GENERATOR START` und `--- ||| GENERATOR END` verloren!

Unterreferenzierte Dhall-Dateien werden nicht automatisch mitgeprüft. Eine Rekursion findet also **nicht** statt. D.h. jede Dhall-Datei muss einzeln geprüft/formatiert werden.

Wir nutzen folgende Kommandos zum Formatieren:

```
CFG="/etc/bffh/bffh.dhall" && \  
dhall format < $CFG && \  
dhall format < $CFG > $CFG.bup && \  
rm $CFG && \  
mv $CFG.bup $CFG
```

Es können jedoch auch die offiziellen Flags verwendet werden. Siehe:

Usage: dhall format [--inplace FILE] [--check]

Standard code formatter for the Dhall language

Available options:

--inplace FILE	Modify the specified file in-place
--check	Only check if the input is formatted
-h,--help	Show this help text

#nur formatieren

```
dhall format --inplace /etc/bffh/bffh.dhall --check
```

formatieren und überschreiben

```
dhall format --inplace /etc/bffh/bffh.dhall
```

Siehe auch [Konfiguration von BFFH auf Synax und Inhalt prüfen](#) für eine BFFH-spezifische Kontrolle der Daten.

Syntax-Highlighting: Dhall-Dateien schöner bearbeiten

Für verschiedene Editoren gibt es Highlighter-Plugins, dabei unter anderem für Vim, VS Code und Emacs. Siehe [hier](#). Eine Referenz findet sich auch in <https://docs.dhall-lang.org/howtos/Text-Editor-Configuration.html>.

Für Vim funktioniert das zum Beispiel so:

```
curl -fLo ~/.vim/autoload/plug.vim --create-dirs https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```

```
vim ~/.vimrc
```

```
call plug#begin()  
Plug 'vmchale/dhall-vim'  
call plug#end()
```

Beim nächsten Öffnen des Editors führen wir das Kommando `:PlugInstall` aus, installieren damit das Plugin und öffnen dann beliebige Dhall-Dateien und sehen, dass uns Syntax-Highlighting nun zur Verfügung steht.

Damit die Datei als Dhall erkannt wird, muss die Dateierweiterung *.dhall lauten.

```

7 db_path = "/opt/fabinfra/bffh-data/bffh",
8 auditlog_path = "/opt/fabinfra/bffh-data/bffh.audit",
9 actor_connections =
10 [
11     {
12         _1 = "Testmachine",
13         _2 = "Actor"
14     }
15 ],
16 actors =
17 {
18     Actor =
19     {
20         module = "Shelly",
21         params = {}
22     }
23 },
24 init_connections = [
25     {
26         _1 = "Initiator",
27         _2 = "Testmachine"
28     }
29 ],
30 initiators =
31 {
32     Initiator =
33     {
34         module = "TCP-Listen",
35         params = {}
36     }
37 },
38 listens =
39 [
40     {
41         address = "127.0.0.1"
42     }
43 ],
44 machines = {},
45 roles = {}

```

Beispiel-Screenshot für Vim

Die Konfiguration von BFFH

Den FabAccess Server zu konfigurieren ist zweifellos eine der wichtigsten Aufgaben und erfordert die Auseinandersetzung mit einigen Konzepten.

Bei jeder Änderung der Konfigurationsdatei muss BFFH neugestartet werden, um die Änderungen zu übernehmen!

Übersicht über alle Einstellungen

In der Config gespeichert werden grundlegende, überwiegend **pflichtmäßig** anzugebende Informationen zu ...

- allgemeine Einstellungen (z.B. Interface, Port, Zertifikate, Audit, Datenbank, etc.)
 - `listens`
 - `db_path`
 - `certfile`
 - `keyfile`
 - `tls_min_version` (optional)

- `ciphers` (optional)
- `mqtt_url`
- `auditlog_path`
- Rollen, Berechtigungen, Ressourcen (Maschinen)
 - `roles`
 - `machines`
- Aktoren, Initiatoren und ihre Mappings
 - `actors`
 - `initiators`
 - `actor_connections`
 - `init_connections`
- Föderation & FabFire
 - `spacename`
 - `instanceurl`

Allgemeine Einstellungen

`listens::List`

Enthält eine Liste aus Einträgen mit einer Adresse (`address::String` (notwendig)), auf die BFFH bei der Verbindung für die API hört sowie einer Portangabe (`port::Integer` (optional)). Die Adresse kann eine IPv4, IPv6 oder ein Hostname sein. Wird der Port nicht angegeben, dann wird der Standardport verwendet. Dieser Port lautet für BFFH `59661` und dabei handelt es sich um einen dynamischen/privaten Port, der auch ohne Root-Zugriff genutzt werden kann (siehe auch "registrierte Ports").

Es ist möglich mehrere Einträge (verschiedene Adressen und Ports) anzugeben. BFFH versucht alle Verbindungen zu etablieren. BFFH kann also z.B. auch gleichzeitig auf `0.0.0.0:59661` und `0.0.0.0:5961` lauschen.

Achtung: Wenn BFFH keinen Port für die angegebene Kombination binden kann, wird ein Fehler protokolliert, aber mit den übrigen Ports fortgefahren.

Beispiele für `listens`:

```
listens = [
  { address = "127.0.0.1", port = Some 59661 },
  { address = ":::1", port = 59661 },
  { address = "steak.fritz.box", port = 59661 } ]
```

mqtt_url::String

Enthält die Adresse des MQTT-Servers, mit dem sich BFFH verbindet.

Die Adresse hat das Format `<protocol>://[user]:[password]@<server>:[port]`

- `protocol` (notwendig) - kann eins der folgenden Werte annehmen: `mqtt`, `tcp`, `mqtt`, `ssl`
- `user` (optional)
- `password` (optional)
- `server` (notwendig) - kann IP-Adresse oder Hostname sein
- `port` (optional) - Der Standardport ist `1883`

Beispiele für mqtt_url:

```
mqtt_url = "tcp://localhost:1883"
```

```
mqtt_url = "mqtt://user:password@server.tld:port"
```

Achtung: Der MQTT-Server muss laufen und erreichbar sein, sonst startet BFFH nicht. Wird die MQTT-Verbindung getrennt, während BFFH läuft, stürzt BFFH nicht ab und protokolliert stattdessen die Verbindungsfehler in die Log-Ausgabe.

certfile::String und keyfile::String

Unsere Kommunikation über Cap'n Proto erfolgt grundsätzlich verschlüsselt. Dafür benötigen wir ein TLS-Zertifikat. Details, wie man ein TLS-Zertifikat generiert: Server - Anleitung zum selber kompilieren.

BFFH benötigt ein PEM-kodierte Zertifikat und den zugehörigen Schlüssel als zwei separate Dateien. BFFH verwendet TLS standardmäßig und ausschließlich in Version 1.2 bzw 1.3. Dabei implementiert es die Rust-Bibliothek rustls im Code.

Beispiel für certfile und keyfile:

```
certfile = "/etc/bffh/certs/bffh.crt"
```

```
keyfile = "/etc/bffh/certs/bffh.key"
```

`ciphers`::String (optional)

Es kann eine Cipher Suite vorgegeben werden, die genutzt werden soll.

Wir empfehlen diesen Wert nicht zu vergeben und beim Standard zu belassen (standardmäßig wird diese Option in `bffh.dhall` nicht gesetzt).

Die Definition erfolgt als String, nicht als List und kann deshalb aktuell nur eine einzelne Cipher Suite enthalten. Siehe Issue 109.

Beispiel für `ciphers`:

```
ciphers = "TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256"
```

Mögliche Cipher Suites sind:

- TLS v1.2 + TLS v1.3
 - `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`
 - `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`
 - `TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256`
 - `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`
 - `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384`
 - `TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256`
- TLS v.1.3 only
 - `TLS13_AES_128_GCM_SHA256`
 - `TLS13_AES_256_GCM_SHA384`
 - `TLS13_CHACHA20_POLY1305_SHA256`

`tls_min_version`::String (optional)

Die TLS Version von BFFH kann erzwungen werden.

Wir empfehlen diesen Wert nicht zu vergeben und beim Standard zu belassen (standardmäßig wird diese Option in `bffh.dhall` nicht gesetzt).

Mögliche Werte:

- `tls12` für TLS v1.2

- `tls13` für TLS v1.3

Beispiel für `tls_min_version`:

```
tls_min_version = "tls12"
```

`protocols`::List (optional)

Gibt an, welche Protokolle durch BFFH genutzt werden dürfen.

Diese Einstellung kann gesetzt werden, hat aber keine Auswirkungen und wird im Code aktuell nicht genutzt. Sie wird nur zur Vollständigkeit hier gelistet. Es ist nicht bekannt, welche Protokolle unterstützt werden sollen!

Beispiel für `protocols`:

```
protocols = ["tcp"],
```

`db_path`::String

Enthält den Pfad für die interne Datenbank auf LMDB-Basis, die BFFH verwendet. BFFH wird beim Start zwei Dateien erstellen, falls nicht bereits existent: `/var/lib/bffh/bffh.db` und `/var/lib/bffh/bffh.db-lock`. BFFH erstellt selber keine fehlenden Verzeichnisse. Es sollte sichergestellt werden, dass BFFH Schreibzugriff auf das entsprechende Verzeichnis hat.

Die interne Datenbank wird im Produktivbetrieb nicht riesig (nur einige Kilobyte bis wenige Megabyte). In ihr werden die aktuellen Ressourcenzustände (states) gespeichert (welche Ressource ist z.B. gerade durch einen Nutzer `INUSE`) und welche Nutzer mit Rollen, Passwörtern und Cardkeys zugreifen dürfen. Diese interne Datenbank kann exportiert werden (Benutzerdaten können als `*.toml` Datei ausgegeben werden oder die ganze Datenbank inklusive Zuständen kann exportiert werden). Wir empfehlen die Ablage im Standardverzeichnis `/var/lib/bffh/`.

Beispiel für `db_path`:

```
db_path = "/var/lib/bffh/bffh.db"
```

BFFH Datenbank "Werkszustand" wiederherstellen:

Die Datenbankdatei und die Lock-Datei können prinzipiell nach Beenden des Dienstes gelöscht werden, jedoch gehen dabei die gespeicherten aktuellen Ressourcenzustände und Nutzer verloren. Die Nutzer können über eine **aktuelle** `users.toml` wieder importiert werden. Dass die Ressourcenzustände verloren gehen, ist je nach Werkstattsetup u.U. einfach zu verkraften. Entsprechend kann damit ein Reset durchgeführt werden. Ggf. sollten alte Audit-Einträge (siehe `auditlog_path`) ebenso bereinigt werden.

Vor jedem Update / Recompile unbedingt beachten:

Die Datenbank speichert ihre Daten nicht im Klartext. Die als Datei erzeugte Datenbank `bffh.db` ist abhängig vom System, wo sie angelegt wurde. Jeder Recompile oder Umzug auf einen anderen Host bzw. Architektur macht die Datenbank unbrauchbar. Deshalb muss auch vor jedem Update eine Sicherung erstellt werden, die jedoch re-importiert werden kann.

Bei Nichtbeachtung droht Verlust der angelegten Benutzerdatenbank und den gespeicherten Ressourcenzuständen.

Die automatisch beim Start erstellte Lock-Datei `/var/lib/bffh/bffh.db-lock` wird nach dem Beenden von BFFH nicht automatisch gelöscht, kann aber manuell immer dann entfernt werden, so lange BFFH nicht läuft. Während des Betriebs sollte die Lock-Datei nicht gelöscht werden, da es sonst zu Korruption in Lese- und Schreibprozessen kommen kann.

`auditlog_path`::String

Pfadangabe zur Auditdatei, die BFFH im laufenden Prozess schreibt. Die Ausgabe erfolgt im JSON-Format. Siehe Audit Log (Revisionsprotokoll) für Details zum Audit. Die Datei kann zum Beispiel mit Tools wie Grafana Loki ausgelesen werden. Wir empfehlen deshalb den Pfad so zu definieren, dass entsprechende Dienste korrekte Leseberechtigungen auf die Datei haben. Der bei Linux-Systemen verwendete Standardpfad for Log-Dateien ist per se `/var/log/`.

Beispiel für `auditlog_path`:

```
auditlog_path = "/var/log/bffh/audit.json"
```

Falls BFFH wegen einem Log-Berechtigungsfehler nicht startet, kann folgender Trick angewendet werden.

Error:

× audit log failed

↳ Permission denied (os error 13)

Lösung: die fehlende Datei neu und leer erzeugen und manuell die passende Berechtigungen setzen:

```
sudo touch /var/log/bffh/audit.json
sudo chown bffh:root /var/log/bffh/audit.json
sudo chmod 750 /var/log/bffh/audit.json
```

Alternativ kann auch mit `setfacl` eine ACL angelegt werden. Hierzu gibt es ein nettes Tool unter <https://techgeeks.io/unix/acl-generator>.

```
sudo setfacl -R -m u:bffh:rw-,d:u:bffh:rw- /var/log/bffh/audit.json
```

Konfiguration von Rollen, Berechtigungen und Ressourcen

FabAccess verwendet eine Role-Based Access Control (RBAC)-Struktur zur Verwaltung von Berechtigungen. Dabei werden Berechtigungen Ressourcenrollen zugewiesen, und diese Rollen werden dann den Benutzern zugewiesen. Auf diese Weise lässt sich ein komplexes Berechtigungssystem einfach, umfassend und flexibel abbilden. Details finden sich unter RBAC (Benutzerrollen und Berechtigungen).

`roles::Record`

Enthält die Einträge der definierten Rollen. Rollen haben eine Liste von Berechtigungen und können vererbt werden. Die Berechtigung kann ein Platzhalter in der Berechtigungsliste sein. Jede Rolle **muss** eine `Role-ID` haben. Unterhalb des Eintrags `roles` schachteln sich Listen vom Typ `parents::List` (optional) und vom Typ `permissions::List` (notwendig).

Standardberechtigungen

BFFH verfügt über einige Standardberechtigungen, die der Verwaltung und den **Admin**-Rechten zugewiesen werden können. Die lauten exakt:

- `bffh.users.info` - Nutzerliste bekommen und Infos über diese Accounts erhalten
- `bffh.users.manage` - Nutzerliste bekommen und Nutzer verwalten
- `bffh.users.admin` - Globale Administration: Nutzer hinzufügen, löschen, ändern (z.B. Passwort-Reset)

Beispiel für eine Admin-Rolle mit Standardberechtigungen:

```
roles = {  
  Admin = {  
    permissions = [  
      "bffh.users.manage",  
      "bffh.users.info",  
      "bffh.users.admin"  
    ]  
  }  
}
```

Modellieren von Berechtigungen für Ressourcen

Allgemeines Schema

`space.type.category.permission.model`

Administrator

`space.machines.printers.*`

Offene Berechtigung

`space.machines.printers.read.*`

Allgemeine Erläuterungen zum Pfadformat und Platzhaltern (*, +)

BFFH verwendet eine pfadähnliche Zeichenkette als Erlaubnisformat, getrennt durch einen `.` Punkt. So besteht zum Beispiel `this.is.a.permission` aus den Teilen `this`, `is`, `a` und `permission`. Bei der Anforderung von Berechtigungen, z. B. in Ressourcen, muss immer eine genaue Berechtigung angegeben werden, also z. B. `test.write`.

Bei der Erteilung von Berechtigungen, z. B. in Rollen, können entweder eine genaue Berechtigung angegeben oder die beiden Platzhalter `*` und `+` verwendet werden. Diese

Wildcards verhalten sich ähnlich wie Regex- oder Bash-Wildcards:

- `*` gewährt alle Berechtigungen in diesem Teilbaum. So wird `perms.read.*` für jedes von passen:
 - `perms.read`
 - `perms.read.machineA`
 - `perms.read.machineB`
 - `perms.read.machineC.manage`
- `+` gewährt alle Berechtigungen unter des Wertes. So wird `perms.read.+` für jedes von passen:
 - `perms.read.machineA`
 - `perms.read.machineB`
 - `perms.read.machineC.manage`
 - **aber nicht** `perms.read`

Wildcards sind wahrscheinlich am nützlichsten, um Ressourcen zu gruppieren, z.B. 3D-Drucker und eine Bandsäge:

1. Write (schreiben) Berechtigungen

- `machines.printers.write.prusa.sl1`
- `machines.printers.write.prusa.i3`
- `machines.printers.write.anycubic`
- `machines.bandsaws.write.bandsaw1`

2. Manage (verwalten) Berechtigungen

- `machines.printers.manage.prusa.sl1`
- `machines.printers.manage.prusa.i3`
- `machines.printers.manage.anycubic`
- `machines.bandsaws.manage.bandsaw1`

3. Admin Berechtigungen

- `machines.printers`
 - Für alle Drucker
- `machines.bandsaws`
 - Für alle Bandsägen

Dann erteilen wir den Rollen die entsprechenden Rechte:

- Nutze beliebige 3D-Drucker:
 - `machines.printers.write.+`
- Erlaube nur die Nutzung "billiger" Drucker:
 - `machines.printers.write.anycubic.*`
 - `machines.printers.write.prusa.i3`
- Erlaube das Verwalten der Drucker:
 - `machines.printers.+`

- Erlaubte das Administrieren aller Drucker:
 - `machines.printers.*`

Auf diese Weise klappt es trotzdem mit der Aufteilung, wenn später ein weitere Anycubic Drucker gekauft wird:

- `machines.printers.write.anycubic.i3`
- `machines.printers.write.anycubic.megax`

Beispiel für verschiedene Rollen mit Berechtigungen:

- **Wildcard-Rolle** `testrole`
- **"normale" Rolle** `anotherrole`
- **Rolle** `somerole` **mit vererbter Rolle** `testparent` :

```
roles = {
  testrole = {
    permissions = [ "lab.some.*" ]
  },
  somerole = {
    parents = [ "testparent" ],
    permissions = [ "lab.some.admin" ]
  },
  testparent = {
    permissions = [
      "lab.some.write",
      "lab.some.read",
      "lab.some.disclose" ]
  },
  anotherrole = {
    permissions = [
      "lab.some.disclose.*",
      "lab.some.read.*",
      "lab.some.write.*" ]
  }
}
```

`machines::Record`

Enthält die Einträge der definierten Ressourcen. Jede Ressource **muss** eine `Resource-ID` haben, um in anderen Teilen dieser Konfiguration oder über die API auf die Ressource verweisen zu können. Und jede Ressource **muss** einen Namen haben. In FabAccess ist die Ressource ein wichtiger Schlüsselbegriff und steht für Dinge wie Türen, Schließfächer, Geräte oder Maschine. Kurzum: Dinge, die sich ein- und ausschalten lassen sollen.

Allgemeine Ressourceninformationen

Um weitere Informationen über die Ressource bereitzustellen, können Namen, Kategorien, Beschreibungen oder ein externer Wiki-Link eingefügt werden:

- `name` ::String (notwendig) - ein Klarname, zum Beispiel ein Spitzname für eine Maschine oder eine Modellbezeichnung (Unicode-Support)
- `description` ::String (optional) - eine allgemeine, kurze Beschreibung der Ressource (Unicode-Support)
- `wiki` ::String (optional) - Link zu einer Wiki- bzw. Dokumentationsseite
- `category` ::String (optional) - eine beliebige Kategorie, z.B. "Holzwerkstatt" oder "Handgerät, elektrisch" (Unicode-Support)
- `producible` ::Boolean (optional) - definiert, ob die Ressource ein Türschloss ist - zum Beispiel an einem Werkzeugschrank, Schließfach oder Eingangstür (ab `v0.4.4`).

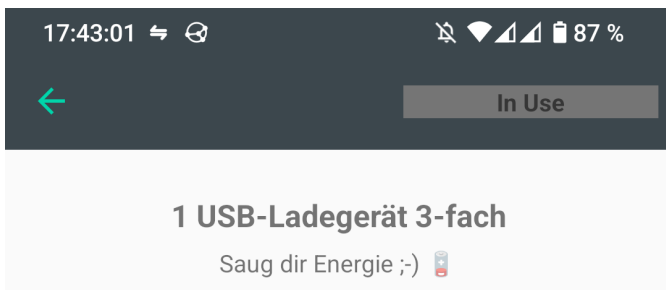
Dieses Attribut schaltet im Borepin Client zwei Buttons frei, die sonst nicht sichtbar sind:

- `UNLOCK` öffnet bzw. entsperrt die Ressource, ohne sie für andere zu blockieren und ohne sie zurückgeben zu müssen
- `IDENTIFY` löst eine einfache Aktion aus wie z.B. das Aufleuchten einer LED an der Ressource, um das Fach bzw. die Ressource optisch leichter zu identifizieren

Was heißt Unicode-Support?

Durch diese Eigenschaft können wir neben einfachen Bezeichnungen auch beliebige Sonderzeichen einfügen. Das erlaubt unter anderem die Verwendung von Emojis, die innerhalb der Client-Anwendung visuell unterstützen können. Dafür kann ein beliebiger Emoji-Picker genutzt werden, zum Beispiel <https://www.freetool.dev/emoji-picker>. D.h. wir können z.B.

`description = "Saug dir Energie ;-) 🧹"`, konfigurieren. Das sieht dann in Borepin so aus:



Zugewiesene Berechtigungen für Ressourcen

Die Ressourcen haben verschiedene Berechtigungsstufen, mit denen interagiert werden kann und welche sich in den Records in `roles` widerspiegeln:

- `disclose`::String (notwendig) - Offenlegen: Der Benutzer kann die Ressource in der Übersichtsliste sehen (entdecken), jedoch keine Details zu ihr aufrufen (Detailansicht erfordert `read` Berechtigung)
- `read`::String (notwendig) - Lesen: Der Benutzer kann allgemeine Informationen über die Ressource und ihren Zustand lesen, sowie z.B. den Wiki Link anklicken
- `write`::String (notwendig) - Schreiben: Der Benutzer kann die Ressource bedienen und dabei folgende Zustände verändern:
 - Benutzen (`InUse`) bzw. Zurückgeben (`Free`, auch als GIVEBACK in der Client App bezeichnet)
 - Reservieren (`Reserve`)
- `manage`::String (notwendig) - Verwalten: Der Benutzer kann als Manager sehen, wer die Maschine aktuell benutzt und wer sie zuletzt benutzt hat. Außerdem können Manager mit dem System interagieren und dabei folgende Zustände verändern:
 - Freigeben erzwingen (force `Free`)
 - Blockieren (`Blocked`)
 - Deaktivieren (`Disabled`)
 - Überprüfen (`ToCheck`)
 - Transferieren (`totakeover`)

Tipp: Wenn einer Ressource die `read` Berechtigung, aber nicht `disclose` Berechtigung gegeben wird, dann taucht die Ressource nicht in der Übersichtsliste auf. Sie kann auf diese Weise nur noch direkt angesprochen werden, indem ein QR-Code oder ein NFC-Tag gescannt wird. So lässt sich erzwingen, dass ein Nutzer zur Maschine lokal hingehen muss und die Ressource nicht von weiter weg bedienen kann (es sei denn der Benutzer klonst den QR-Code oder NFC-Tag und nimmt ihn mit).

Bitte beachten:

- Jeder Ressource **müssen alle** Berechtigungsstufen zugewiesen sein.
- Berechtigungsstufen sind nicht additiv, d.h. ein Benutzer mit der Berechtigung `manage` erhält nicht automatisch `read` oder `write` Berechtigung.
- `disclose` ist im Moment **nicht** vollständig implementiert. Benutzer, die keine `disclose` Berechtigung haben, werden in keiner Weise über diese Ressource informiert und sie wird vor ihnen im Client verborgen. Es ist deshalb am sinnvollsten, wenn `read` und `disclose` die gleiche Berechtigung zugewiesen bekommen, zum Beispiel `disclose = "lab.test.read"`.
- Benutzern ohne `read` wird eine Resource mit Namen, Beschreibung, Kategorie und Wiki angezeigt, aber nicht ihr aktueller Zustand. Der aktuelle Benutzer der Ressource wird nicht offengelegt.

Beispiel einer Ressource mit allen Parametern:

```
machines = {  
  machine_123 = {  
    name = "Testmachine",  
    description = Some "A test machine",  
    wiki = "https://someurl",  
    category = "Testzone",  
    prodable = True,  
    disclose = "lab.some.read",  
    read = "lab.some.read",  
    write = "lab.some.write",  
    manage = "lab.some.admin"  
  }  
}
```

Im Beispiel heißt unser `Resource-ID` `machine_123`.

Konfiguration von Aktoren (actors)

`actors::Record`

Enthält eine Liste von Aktoren. Aktoren werden durch ein Modul `module::String` (notwendig) und einen oder mehrere Parameter `params::Record` definiert. Aktuell **von Haus aus unterstützte Aktoren** (ohne zusätzliche Plugins) sind

Dummy Actor, Shelly Actor und Process Actor.

Konkret nutzbare Aktoren-Beispiele finden sich hier.

Aktoren werden durch ein Modul und einen oder mehrere Parameter definiert. Die Liste kann bzw. **muss** leer sein, wenn keine Aktoren verwendet werden:

Beispiel für leere actors:

```
actors = {=}
```

Sonst kann die Liste einen oder mehrere Aktoren mit ihrem Name (`Actor-ID`) enthalten:

Dummy Actor

Für Testzwecke kann ein interner Dummy-Initiator genutzt werden. Der „Dummy“-Initiator versucht alle paar Sekunden, einen Rechner als den angegebenen Benutzer zu verwenden und zurückzugeben. Er ist gut geeignet, um das System zu testen, führt aber zu Spam im Log und ist daher standardmäßig deaktiviert.

Beispiel für Dummy Actor:

```
actors = {  
  Dummy_123 = {  
    module = "Dummy",  
    params = {=}  
  }  
}
```

Im Beispiel heißt unser `Actor-ID` `Dummy_123`.

Shelly Actor

Dieser Aktor verbindet BFFH über einen MQTT-Server mit einem Shelly Gerät (dafür wird intern die MQTT-Bibliothek `rumqtmc` verwendet). Der Parameter `topic ::String` (optional) des Shelly kann auf das Shelly-spezifische MQTT-Topic gesetzt werden. Wird kein `topic` definiert, dann verwendet der Actor automatisch die definierte `Actor-ID`.

Anleitung zum Auffinden des Shelly Topic

Beispiel für Shelly Actor:

```
actors = {  
  Shelly_123 = {  
    module = "Shelly",  
    params = {  
      topic = "shellyplug-s-123456"  
    }  
  }  
}
```

Im Beispiel heißt unser `Actor-ID` `Shelly_123`.

Process Actor

Dieser Aktor ermöglicht es, eigene Prozesse (z.B. Python, Bash, Perl, Java ...) mit dem BFFH Server zu verbinden.

`cmd::String` (notwendig) - Pfad der ausführbaren Datei

`args::String` (optional) - Argumente der ausführbaren Datei. Hier übergebene Argumente werden durch Leerzeichen getrennt, so dass in unserem Beispiel 5 separate Argumente übergeben werden.

Beispiel für Process Actor:

```
actors = {  
  Bash_123 = {  
    module = "Process",  
    params = {  
      cmd = "./examples/actor.sh",  
      args = "your ad could be here"  
    }  
  }  
}
```

Im Beispiel heißt unser `Actor-ID` `Bash_123`.

Wichtige Info: BFFH baut aus den Argumenten `cmd` und `args` einen vollständigen Befehl, der ausgeführt wird und fügt automatisch zwei weitere Argumente am Ende hinzu - nämlich den zu erzielenden Status `state` (z.B. `INUSE` oder `GIVEBACK`) und den

Benutzername `userid` . Der vollständige Befehl, der als Prozess ausgeführt wird, lautet folglich `$ ${cmd} ${args} ${state} ${userid}` .

`actor_connections::List`

Verknüpfung (Mapping) von Ressourcen mit Aktoren. Einer Ressource können mehrere Aktoren zugewiesen werden, aber ein Aktor kann nur einer Ressource zugewiesen werden.

Das Mapping `actor_connections` wird als Liste von Einträgen gespeichert. Die Parameter:

- `machine::String` (notwendig) - die `Resource-ID`
- `actor::String` (notwendig) - die `Actor-ID`

Achtung:

Die Reihenfolge ist wichtig. An erster Stelle kommt `machine` , dann `actor` !

Außerdem darf die Definition `actor_connections::List` in `bffh.dhall` erst **nach** `actors::Record` und **nach** `machines::Record` kommen!

Beispiel für leere actor_connections:

```
actor_connections = [] : List { machine : Text, initiator : Text },
```

Beispiel für init_connections mit drei Mappings:

```
actor_connections = [  
  { machine = "Testmachine", actor = "Shelly_123" },  
  { machine = "Another", actor = "Bash_123" },  
  { machine = "Yetmore", actor = "Bash_234" } ]
```

Konfiguration von Initiatoren (initiators)

Initiatoren werden fast genauso konfiguriert wie Aktoren. Es gibt aktuell die beiden Initiatorentypen Dummy Initiator und Process Initiator. Konkret nutzbare Initiatoren-Beispiele finden sich [hier](#).

`initiators::Record`

Enthält eine Liste von Initiatoren. Initiatoren werden durch ein Modul und einen oder mehrere Parameter definiert. Die Liste kann bzw. **muss** leer sein, wenn keine Initiatoren verwendet werden:

Beispiel für leere initiators:

```
initiators = {}
```

Sonst kann die Liste einen oder mehrere Initiatoren mit ihrem Name (`Initiator-ID`) enthalten:

Dummy Initiator

Für Testzwecke kann ein interner Dummy-Initiator genutzt werden:

Der „Dummy“-Initiator versucht alle paar Sekunden, einen Rechner als den angegebenen Benutzer zu verwenden und zurückzugeben. Er ist gut geeignet, um das System zu testen, führt aber zu Spam im Log und ist daher standardmäßig deaktiviert.

Die Initiator-Parameter:

`uid::String` (notwendig) - Nutzernamen

Beispiel für Dummy Initiator:

```
initiators = {  
  Initiator_123 = {  
    module = "Dummy",  
    params = {  
      uid = "Testuser"  
    }  
  }  
}
```

Process Initiator

Dieser Initiator vom `module::String` Typ **Process** ermöglicht es, eigene Prozesse (z.B. Python, Bash, Perl, Java ...) mit dem BFFH Server zu verbinden.

Das Mapping `initiator_connections` wird als Liste von Einträgen gespeichert. Die Parameter:

`cmd::String` (notwendig) - Pfad der ausführbaren Datei

`args::String` (optional) - Argumente der ausführbaren Datei. Hier übergebene Argumente werden durch Leerzeichen getrennt, so dass in unserem Beispiel 5 separate Argumente übergeben werden.

Beispiel für Process Initiator:

```
initiators = {  
    Bash_567 = {  
        module = "Process",  
        params = {  
            cmd = "./examples/init.py",  
            args = "your ad could be here"  
        }  
    }  
}
```

Im Beispiel heißt unser `Initiator-ID` `Bash_567`.

Wichtige Info: BFFH baut aus den Argumenten `cmd` und `args` einen vollständigen Befehl, der ausgeführt wird und fügt automatisch zwei weitere Argumente am Ende hinzu - nämlich den zu erzielenden Status `state` (z.B. `INUSE` oder `GIVEBACK`) und den Benutzernamen `userid`. Der vollständige Befehl, der als Prozess ausgeführt wird, lautet folglich `$ ${cmd} ${args} ${state} ${userid}`.

`init_connections::List`

Verknüpfung (Mapping) von Ressourcen mit Initiatoren. Einer Ressource können mehrere Initiatoren zugewiesen werden, aber ein Initiator kann nur einer Ressource zugewiesen werden.

Das Mapping `init_connections` wird als Liste von Einträgen gespeichert. Die Parameter:

- `initiator::String` (notwendig) - die `Initiator-ID`
- `actor::String` (notwendig) - die `Actor-ID`

Achtung: Die Reihenfolge ist wichtig. An erster Stelle kommt `machine`, dann `initiator` !

Außerdem darf die Definition `init_connections :: List in bffh.dhall` erst **nach** `initiators :: Record` und **nach** `machines :: Record` kommen!

Beispiel für leere `init_connections`:

```
init_connections = [] : List { machine : Text, initiator : Text }
```

Beispiel für `init_connections` mit einzelnen Mapping:

```
init_connections = [  
  { machine = "Testmaschine", initiator = "Initiator_123" }  
]
```

FabFire und Föderation

`spacename` :: String

Der Name des Spaces (die offene Werkstatt, das FabLab, der HackerSpace, etc.) wird im URN-Schema `urn:fabaccess:lab:{spacename}` verwendet. Wird er nicht definiert, wird der Wert "generic" vergeben. Diese Angaben benötigen wir für QR-Codes von Ressourcen oder für DESFire Karten zur Nutzung von FabFire.

Beispiel für `spacename`:

```
spacename = "FabAccess DemoSpace"
```

`instanceurl` :: String

Wird für eine allgemeine Space Info genutzt und als URN im Code genutzt:

`urn:fabaccess:lab:{spacename}\x00{instanceurl}`. Dieser Wert wird aktuell nicht verwendet, muss jedoch ausgefüllt werden, damit die Konfiguration `bffh.dhall` valide ist! Es wird empfohlen die URL mit dem Protokoll vollständig anzugeben, also mit `http://` oder `https://`.

Beispiel für `instanceurl`:

```
instanceurl = "https://demo.fab-access.org"
```

Hinweise zu unkonfigurierbaren Variablen

Folgende Variablen werden intern von BFFH gesetzt und können aktuell nicht über die `bffh.dhall` verändert werden. Wir führen sie nur zur Vollständigkeit auf:

- `verbosity`
- `filter`
- `format`

Hinweise zur Vergabe von IDs

Wir verwenden in der Konfiguration IDs wie `Resource-ID`, `Actor-ID`, `Initiator-ID` oder `Role-ID`.

Diese IDs müssen folgendes Namensschema einhalten:

- ASCII Format
- Alphanumerisch
- muss mit einem Buchstaben beginnen

Für die Vergabe von IDs ist die Groß- und Kleinschreibung zu beachten! Wir empfehlen generell die Kleinschreibung. So ist zum Beispiel eine Rolle namens `RoleAdmin` ungleich `roleadmin`.

Wir empfehlens außerdem ein konsistentes, leicht nachvollziehbares (logisches) Namenskonzept über alle ID-Typen, wo immer es möglich ist.

IDs müssen eindeutig bzw. einzigartig sein. IDs dürfen also nicht doppelt vergeben werden!

Minimal funktionierende Standardkonfiguration

Wie bereits eingangs erwähnt, werden alle obigen Einstellungen für die Konfiguration benötigt. Für die absolut minimale Startfähigkeit eines BFFH Servers kann folgendes Konfigurationssample verwendet werden.

Vorraussetzungen für die Lauffähigkeit von BFFH:

- Dienst muss sich auf Interface:Port binden lassen
- MQTT-Server muss laufen und erreichbar sein (siehe `mqtt_url::String`).
- Zertifikat und Keyfile müssen existieren
- Datenbankdatei und Auditlog müssen schreibbar sein

```
{
  spacename = "fabaccess.sample.space",
  instanceurl = "https://fabaccess.sample.space",
  listens = [{address = "127.0.0.1"}],
  certfile = "/etc/bffh/certs/bffh.crt",
  keyfile = "/etc/bffh/certs/bffh.key",
  mqtt_url = "mqtt://127.0.0.1:1883",
  db_path = "/var/lib/bffh/bffh.db",
  auditlog_path = "/var/log/bffh/audit.json",
  roles = {=},
  machines = {=},
  actors = {=},
  actor_connections = [] : List { machine : Text, initiator : Text },
  initiators = {=},
  init_connections = [] : List { machine : Text, initiator : Text }
}
```

Ein ebenso funktionales und umfangreicheres Standardsample kann auch per `bffhd --print-default` ausgegeben werden (siehe [Cheat Sheet](#)).

Konfiguration per Config Generator

Ein Großteil der Konfiguration (Ressourcen, Berechtigungen, Aktoren, Aktorenverbindungen, etc.) kann mit einem generischen Arbeitswerkzeug effizient und übersichtlich erzeugt und verwaltet werden. Siehe [Einfache Konfiguration mit dem FabAccess Config Generator](#).

Konfiguration von BFFH auf Synax und Inhalt prüfen

Konfiguration erstellt, aber unsicher, ob sie vollständig und syntaktisch korrekt ist?

Wir können das per `/usr/bin/bffhd --check --config /etc/bffh/bffh.dhall` überprüfen. Siehe [hier](#). Das Kommando kann mit dem `--check` Parameter unabhängig und vor dem Neustart von BFFH ausgeführt werden und verhindert so die Unverfügbarkeit des Service bei Fehlkonfiguration (alternativ kann auch ein Staging-System für derartige Tests angelegt werden). Etwaige Fehlkonfigurationen werden auch erkannt, da geprüft wird, ob wichtige Konfigurationsschlüssel fehlen, unvollständig oder falsch sind.

Siehe auch [Dhall-Dateien parsen, prüfen und formatieren](#) für die reine Überprüfung des Syntax und das automatische Umformatieren (Prettyprint).

Ein letzter Blick in die von BFFH geladene Config kann auch über unsere systemd Journaleinträge gemacht werden, falls wir BFFH als Service installiert haben, oder aber über die Log-Umgebungsvariable `BFFH_LOG`. Wir sehen damit, wie BFFH die Dhall nach dem Parsen vollständig auflöst und dass Log-Variablen beispielsweise auch gesetzt werden, welche jedoch nicht über `bffh.dhall` konfigurierbar sind, sondern über Umgebungsvariablen.

```
sudo systemctl restart bffh.service; sudo journalctl -n2000 -e -u bffh.service
```

```
Jan 02 22:31:07 fabaccess systemd[1]: Starting bffh.service - FabAccess BFFH Service...
Jan 02 22:31:07 fabaccess bffhd[34844]: Config {
Jan 02 22:31:07 fabaccess bffhd[34844]:   listens: [
Jan 02 22:31:07 fabaccess bffhd[34844]:     Listen {
Jan 02 22:31:07 fabaccess bffhd[34844]:       address: "0.0.0.0",
Jan 02 22:31:07 fabaccess bffhd[34844]:       port: Some(
Jan 02 22:31:07 fabaccess bffhd[34844]:         59661,
Jan 02 22:31:07 fabaccess bffhd[34844]:       ),
Jan 02 22:31:07 fabaccess bffhd[34844]:     },
Jan 02 22:31:07 fabaccess bffhd[34844]:   ],
Jan 02 22:31:07 fabaccess bffhd[34844]:   machines: {},
Jan 02 22:31:07 fabaccess bffhd[34844]:   actors: {},
Jan 02 22:31:07 fabaccess bffhd[34844]:   initiators: {},
Jan 02 22:31:07 fabaccess bffhd[34844]:   mqtt_url: "mqtt://fabinfra101:fablocal@0.0.0.0:1883",
Jan 02 22:31:07 fabaccess bffhd[34844]:   actor_connections: [],
Jan 02 22:31:07 fabaccess bffhd[34844]:   init_connections: [],
Jan 02 22:31:07 fabaccess bffhd[34844]:   db_path: "/var/lib/bffh/bffh.db",
Jan 02 22:31:07 fabaccess bffhd[34844]:   auditlog_path: "/var/log/bffh/audit.json",
Jan 02 22:31:07 fabaccess bffhd[34844]:   roles: {},
Jan 02 22:31:07 fabaccess bffhd[34844]:   tlsconfig: TlsListen {
Jan 02 22:31:07 fabaccess bffhd[34844]:     certfile: "/etc/bffh/certs/bffh.crt",
Jan 02 22:31:07 fabaccess bffhd[34844]:     keyfile: "/etc/bffh/certs/bffh.key",
Jan 02 22:31:07 fabaccess bffhd[34844]:     ciphers: None,
Jan 02 22:31:07 fabaccess bffhd[34844]:     tls_min_version: None,
Jan 02 22:31:07 fabaccess bffhd[34844]:     protocols: [],
Jan 02 22:31:07 fabaccess bffhd[34844]:   },
Jan 02 22:31:07 fabaccess bffhd[34844]:   tlskeylog: None,
```

```
Jan 02 22:31:07 fabaccess bffhd[34844]:    verbosity: 0,  
Jan 02 22:31:07 fabaccess bffhd[34844]:    logging: LogConfig {  
Jan 02 22:31:07 fabaccess bffhd[34844]:        filter: None,  
Jan 02 22:31:07 fabaccess bffhd[34844]:        format: "full",  
Jan 02 22:31:07 fabaccess bffhd[34844]:    },  
Jan 02 22:31:07 fabaccess bffhd[34844]:    spacename: "FabAccess Demo Setup",  
Jan 02 22:31:07 fabaccess bffhd[34844]:    instanceurl: "https://docs.fab-access.org",  
Jan 02 22:31:07 fabaccess bffhd[34844]: }  
Jan 02 22:31:07 fabaccess systemd[1]: Started bffh.service - FabAccess BFFH Service.
```

Version #144

Erstellt: 23 Oktober 2024 22:53:35 von Mario Voigt (Stadtfabrikanten e.V.)

Zuletzt aktualisiert: 17 März 2025 16:33:21 von Mario Voigt (Stadtfabrikanten e.V.)