Grundkonzepte

Ein Einstieg in die Grundkonzepte von FabAccess.

Die Anforderungen an FabAccess sind im Laufe der Jahre rasant gewachsen. Ursprünglich sollte nur eine Drehbank mit Strom versorgt werden, nun können jedoch auch Türen oder Schließfächer verwaltet werden. Um diesen vielfältigen Anforderungen gerecht zu werden, haben wir einige Konzepte erarbeitet, mit denen wir den Prozess zum Freischalten dieser Ressourcen vereinheitlichen und für alle verfügbar machen wollen.

Bitte beachte: noch nicht alle hier festgehaltenen Grundkonzepte sind in FabAccess implementiert. Hier bedarf es Entwicklerarbeit.

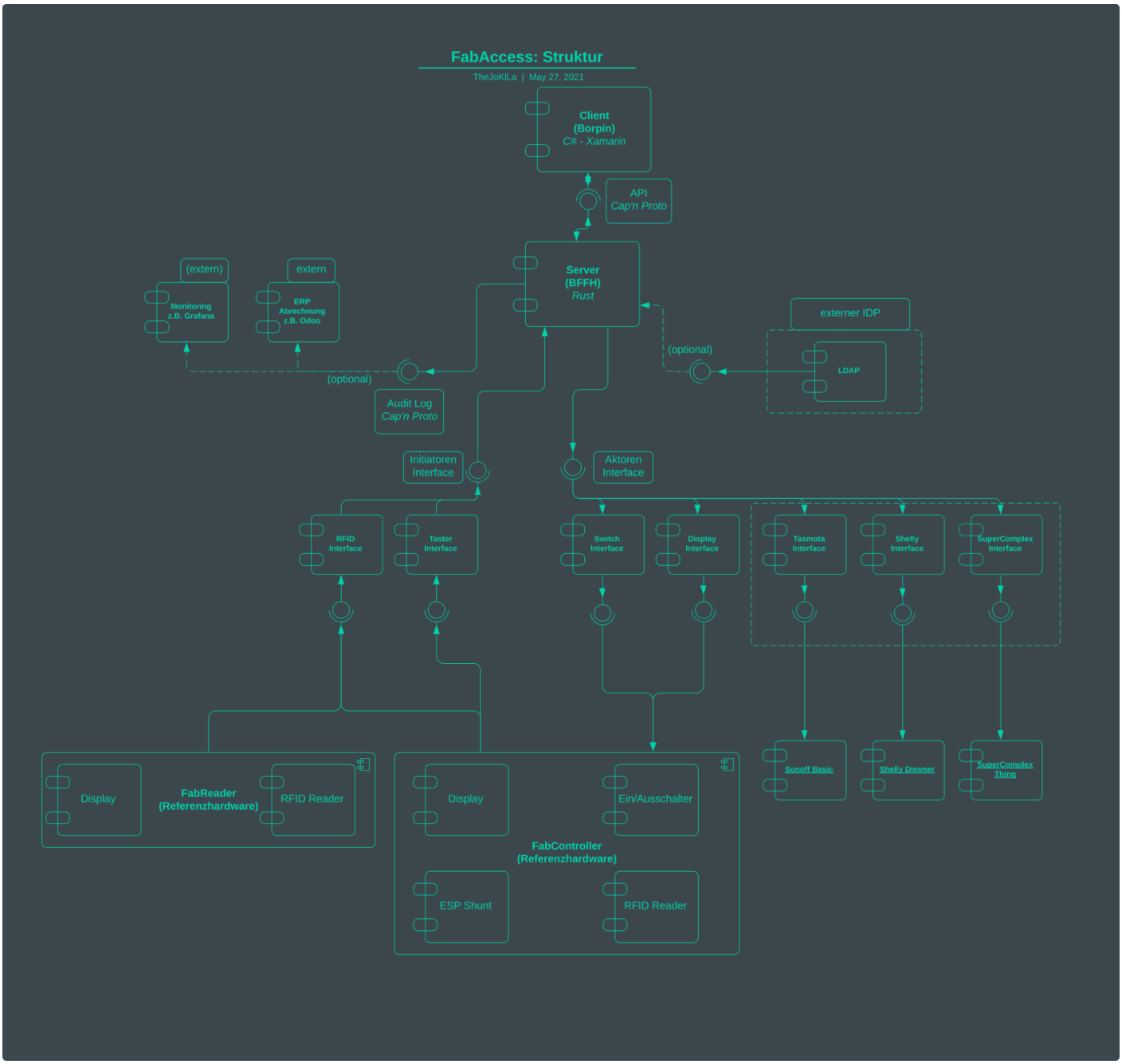
Die Implementierungen von FabAccess resultieren aus den Konzepten. Dabei müssen oft weitere Annahmen getroffen werden, wie genau die Implementierung aussehen soll. Für all diese Fälle werden hier die Entscheidungen festgehalten und möglichst nachvollziehbar erklärt, warum die Entscheidung so getroffen wurde.

Die hier aufgelisteten Konzepte haben viel mit Softwareimplementierung zu tun. Deshalb verwenden wir in den Titeln englische Begriffe, erläutern aber an geeigneten Stellen die deutsche Interpretation.

- Measurements (Messwerte)
- RBAC (Benutzerrollen und Berechtigungen)
- Plugins
- Projekte
- Terminals
- Externe Authentifikation
- Nutzerverwaltung
- Sensoren
- Zustände (Traits)
- URL und URN

Allgemeine Struktur

Strukturdiagramm von Joseph Langosch, Stand 27.05.2021, es zeigt die grundlegenden Komponenten und deren Interaktionen



FabAccess Server-Client Konzept

Stream Initiierung

Bei einer Sitzung gibt es zwei Parteien: Die einleitende Stelle und die empfangende Stelle. Diese Terminologie bezieht sich nicht auf den Informationsfluss, sondern vielmehr auf die Seite, die eine Verbindung eröffnet bzw. auf die Seite, die auf Verbindungsversuche wartet. In dem derzeit angedachten Anwendungsfall ist die initiierende Instanz ...

- a) ein Client (d.h. ein interaktives oder Batch/automatisiertes Programm), der versucht, auf die eine oder andere Weise mit einem Server zu interagieren,
- b) ein Server, der versucht, Informationen mit/von einem anderen Server auszutauschen/anzufordern (d.h. Föderation). Die empfangende Einheit ist jedoch bereits ein Server.

Außerdem ist die Anzahl und Art der Clients vielfältiger und weniger aktuell als die der Server. Daraus ziehen wir folgende Schlüsse:

- Es ist wahrscheinlicher, dass Clients eine veraltete Version des Kommunikationsprotokolls implementieren.
- Der Ort für Rückwärtskompatibilität sollten die Server sein.
- Daher sollte der Client (auslösende Einheit) zuerst die erwartete API-Version senden, auf deren Grundlage der Server dann entscheidet, mit welcher API-Version er antworten wird.

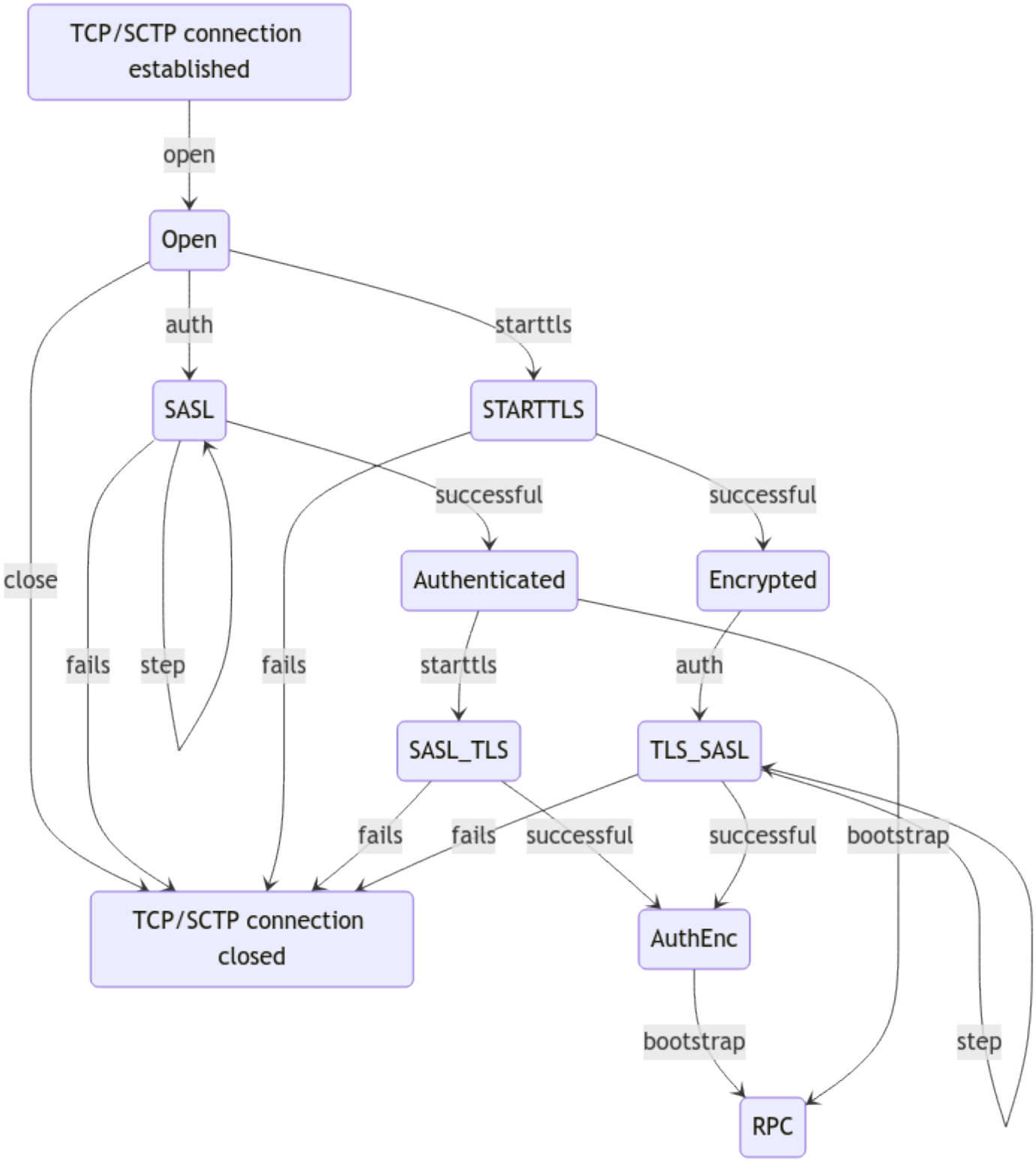
Stream Negotiation

Da die empfangende Stelle für die von ihr kontrollierten Ressourcen verantwortlich ist, stellt sie Bedingungen für die Verbindung entweder als Client oder als föderierender Server. Zumindest muss sich jede einleitende Stelle gegenüber der empfangenden Stelle authentifizieren, bevor sie weitere Aktionen durchführt oder Informationen anfordert. Eine empfangende Stelle kann aber auch andere Merkmale verlangen, z. B. eine Verschlüsselung auf der Transportschicht. Zu diesem Zweck informiert eine empfangende Stelle die einleitende Stelle über Merkmale, die sie von der einleitenden Stelle benötigt, bevor sie weitere Aktionen durchführt, sowie über Merkmale, die freiwillig ausgehandelt werden, aber die Qualität des Datenstroms verbessern können (z. B. Nachrichtenkompression).

Unterschiedliche Bedingungen machen eine Verhandlung erforderlich. Da die Funktionen möglicherweise eine strenge Reihenfolge erfordern (z. B. Verschlüsselung vor

Authentifizierung), muss die Aushandlung in mehreren Schritten erfolgen. Weitere Einschränkungen ergeben sich daraus, dass einige Funktionen erst angeboten werden können, nachdem andere eingerichtet wurden (z. B. SASL-Authentifizierung wird erst nach der Verschlüsselung verfügbar, der EXTERNAL-Mechanismus ist nur für lokale Sockets oder Verbindungen verfügbar, die ein Zertifikat bereitstellen).

Verbindungsstatus

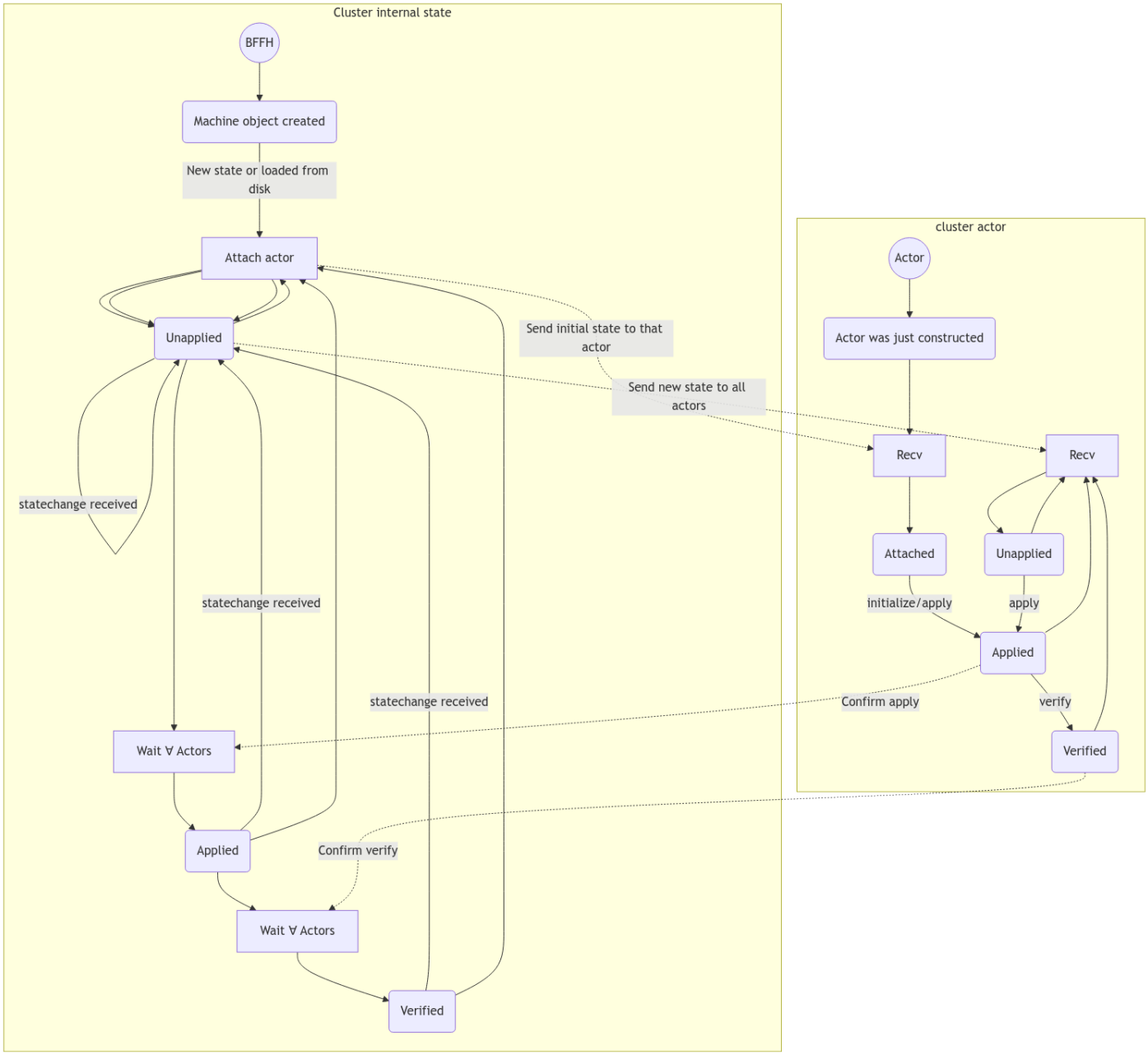


als Mermaid

```
stateDiagram
    state "TCP/SCTP connection established" as Establish
    state "TCP/SCTP connection closed" as Closed
    Open
    SASL
    Authenticated
    STARTTLS
    Encrypted

    Establish --> Open:open

    Open --> Closed:close

    Open --> SASL:auth
    SASL --> SASL:step
    %% Authentication fails
    SASL --> Closed:fails
    %% Authentication succeeds
    SASL --> Authenticated:successful

    Open --> STARTTLS:starttls
    %% TLS wrapping succeeds
    STARTTLS --> Encrypted:successful
    %% TLS wrapping fails
    STARTTLS --> Closed:fails

    Authenticated --> SASL_TLS:starttls
    SASL_TLS --> Closed:fails
    SASL_TLS --> AuthEnc:successful

    Encrypted --> TLS_SASL:auth
    TLS_SASL --> TLS_SASL:step
    TLS_SASL --> Closed:fails
    TLS_SASL --> AuthEnc:successful

    %% Only authenticated connections may open RPC. For "unauth", use the `Anonymous` SASL
```

```
method.
    AuthEnc --> RPC:bootstrap
    Authenticated --> RPC:bootstrap
```

Andere Stati (Cluster Kommunikation BFFH und Aktoren)



als Mermaid

```
graph

    subgraph First[Cluster internal state]
        start((BFFH))
        created(Machine object created)
        start --> created

        created -- New state or loaded from disk --> attach
        attach[Attach actor]
        unapplied(Unapplied)
        applied(Applied)
        verified(Verified)

        wait_apply[Wait ∀ Actors]
        wait_verify[Wait ∀ Actors]

        unapplied --> wait_apply
        wait_apply --> applied
        applied --> wait_verify
        wait_verify --> verified

        applied -- statechange received --> unapplied
        verified -- statechange received --> unapplied
        unapplied -- statechange received --> unapplied

        unapplied --> attach
        attach --> unapplied
        applied --> attach
        attach --> unapplied
        verified --> attach
        attach --> unapplied
    end


    subgraph Second[cluster actor]
        actor_start((Actor))
        actor_fresh(Actor was just constructed)
        actor_start --> actor_fresh
```

```
    actor_attached(Attached)
    actor_unapplied(Unapplied)
    actor_applied(Applied)
    actor_verified(Verified)

    wait_initial[Recv]
    wait_state[Recv]

    actor_fresh --> wait_initial
    wait_initial --> actor_attached

    actor_attached -- initialize/apply --> actor_applied
    actor_unapplied -- apply --> actor_applied
    actor_applied -- verify --> actor_verified

    actor_unapplied --> wait_state
    actor_applied --> wait_state
    actor_verified --> wait_state

    wait_state --> actor_unapplied
end

attach -. Send initial state to that actor .-> wait_initial
unapplied -. Send new state to all actors .-> wait_state
actor_applied -. Confirm apply .-> wait_apply
actor_verified -. Confirm verify .-> wait_verify
```

Hinweis: Die Mermaid-Diagramme sind mit https://mermaid.live gerendert und als PNG exportiert und hier importiert worden, da BookStack keinen integrierten Mermaid Renderer besitzt.

Abhängigkeiten von Ressourcen

FabAccess unterstützt die Verwaltung von Ressourcenabhängigkeiten. Dabei werden automatisch Claims auf Ressourcen an die Nutzer vergeben, die eine Ressource beanspruchen, die von einer anderen abhängt.

Die abhängige Ressource kann zusammen mit der anderen Ressource zurückgegeben werden. Auch die Zustände der Traits werden bei abhängigen Ressourcen berücksichtigt. Auf diese Weise kann verhindert werden, dass Ressourcen wie Absaugungen oder Kühlungen ausgeschaltet werden, wenn die Ressource, die diese benötigt, noch aktiv ist.

Dieses Konzept ist auch als "Primary-Secondary" oder "Master-Slave" bekannt.

Aktoren (Actors) und Initiatoren (Initiators)

Um FabAccess erweiterbar zu halten, basiert die Steuerung externer Geräte wie Wifi-Schaltsteckdosen oder Türschlössern auf einem Aktoren- und Initiatorenkonzept.

Aktoren (Actors)

Aktoren in FabAccess haben die Aufgabe, die digitalen Zustände von Ressourcen in reale Zustände umzusetzen. Vom Server aus werden die Übergänge der Traits (Zustände) an kleine Skripte oder Prozesse weitergegeben, die entsprechend darauf reagieren.

Aktoren erhalten vom Server Mitteilungen über Änderungen an Ressourcen, beispielsweise wenn eine Ressource ausgeliehen wird, und passen dann den realen Zustand der Ressource an. Dadurch wird die Ressource für Nutzer freigeschaltet / eingeschaltet. Darüber hinaus ermöglichen Aktoren die Steuerung zusätzlicher Anzeigen wie Maschinenampeln oder Displays.

Aktoren werden in der Hauptkonfiguration `bffh.dhall` definiert und an Ressourcen (Maschinen) gebunden (Mapping actor_connections).

Initiatoren (Initiators)

Initiatoren sind STDOUT/Prozess-basierte Plugins. Mit Initiatoren werden reale Zustände auf die digitalen Zustände in FabAccess abgebildet. Initiatoren sind Plugins, die den Status einer Ressource aktiv verändern können, also zum Beispiel von `Free` auf `InUse` setzen. Sie fungieren sozusagen als Callbacks. Die Auslösevents (Trigger) sind in der Regel asynchron, von der App entkoppelt und lösen automatisiert aus.

Initiatoren werden in der Hauptkonfiguration `bffh.dhall` definiert und an Ressourcen (Maschinen) gebunden (Mapping init_connections).

Beispiele

Mit Initiatoren können im einfachsten Fall fest definierte Initialzustände an Ressourcen übersendet werden, nachdem BFFH neugestartet wurde - zum Beispiel für einen täglichen "Werkstattreset". So können u.a. bestimmte Dinge automatisch ein-/ausgeschalten bzw. in einen klar definierten Ausgangszustand versetzt werden. Ein Beispiel dafür findet sich in Initiator: Generisches Python-Template für "Process".

Initiatoren ermöglichen auch Zeitschaltungen für die automatische Rückgabe von Ressourcen, wenn sie nicht mehr verwendet werden (z.B. wenn Stromverbrauch für mehrere Minuten unterhalb eines Verbrauchsschwellwerts, dann MQTT-Nachricht versenden und Ressource freigeben). Eine Beispielimplementation findet sich in Initiator: Shelly Timeout.

Türkontakte, die den aktuellen Zustand der Tür übertragen können, können ebenso als Initiator geschrieben werden und BFFH kann dann entsprechend darauf reagieren.

Sammlung von Aktoren und Initiatoren

Eine Sammlung von Initiatoren und Aktoren ist hier zu finden.

Audit Log (Revisionsprotokoll)

Interaktionen und Ergebnisse der Verwendung von Ressourcen werden protokolliert, um sie später auswerten zu können - zum Beispiel im Fehlerfall, in der Schadensaufklärung oder für die Abrechnung der Nutzung (Nutzungsgebühren).

Über das Audit Log File

Das Audit-Log leitet alle Änderungen an Ressourcen vom Server aus weiter. Über selbst geschriebene Plugins kann man diese Informationen nutzen, um z.B. Abrechnungen zu erstellen oder Maschinenzeiten grafisch auszuwerten. Diese Funktion bietet auch Möglichkeiten für eine detaillierte Analyse und Optimierung der Maschinennutzung.

Als zusätzliche Option können eigene Prozesse an die API gebunden werden. Diese ermöglichen einen direkten Zugriff auf die Funktionen des Servers über die API und eröffnen die Möglichkeit für Massenänderungen oder die zeitgesteuerte Zuweisung von Rollen.

Beim Erzeugen von Vorratsdaten ist stets dabei darauf zu achten, nur die minimal notwendigen Daten zu speichern und auszuwerten, um die Privatsphäre der Nutzer zu schützen und das Vertrauen nicht zu verletzen. FabAccess ist hier sehr sparsam und schreibt generisch je Ereignis eine Zeile im JSON-Format. Der Zeitstempel ist dabei im Unix-Format:

```
{"timestamp":<UNIX Zeitstempel>,"machine":"Ressourcen-ID","state":"<Status> <Benutzer>"}
```

Bffh protokolliert Zustandsänderungen in der Audit-Log-Datei (der Pfad wird über `auditlog_path` in der Konfigurationsdatei (*.dhall) festgelegt). Ausschnitt einer Beispiel Log File:

```
{"timestamp":1726239904,"machine":"Fokoos","state":"inuse local_lab_admin"}
{"timestamp":1726239905,"machine":"Fokoos","state":"free"}
{"timestamp":1726239932,"machine":"Ender","state":"inuse local_lab_admin"}
{"timestamp":1726239933,"machine":"Ender","state":"free"}
{"timestamp":1726240081,"machine":"Ender","state":"inuse local_lab_admin"}
{"timestamp":1726240088,"machine":"Ender","state":"free"}
{"timestamp":1726240112,"machine":"Mjolnir","state":"inuse local_lab_admin"}
{"timestamp":1726240122,"machine":"Mjolnir","state":"disabled"}
{"timestamp":1726240125,"machine":"Mjolnir","state":"disabled"}
```

```
{"timestamp":1726240126,"machine":"Mjolnir","state":"blocked local_lab_admin"}
{"timestamp":1726240128,"machine":"Mjolnir","state":"free"}
{"timestamp":1726240132,"machine":"Mjolnir","state":"inuse local_lab_admin"}
{"timestamp":1726240134,"machine":"Mjolnir","state":"free"}
{"timestamp":1726240139,"machine":"BashMachine","state":"inuse local_lab_admin"}
{"timestamp":1726240141,"machine":"BashMachine","state":"free"}
```


Parsing mit JQ

```
sudo apt install jq
```

Allgemeines Parsen

```
jq . /var/log/bffh-audit.json
```


```
{
  "timestamp": 1727110784,
  "machine": "Mjolnir",
  "state": "inuse local_lab_admin"
}
{
  "timestamp": 1729189184,
  "machine": "Mjolnir",
  "state": "free"
}
{
  "timestamp": 1729189186,
  "machine": "Mjolnir",
  "state": "inuse local_lab_admin"
}
```


UTC Timestamps

Parsen mit allgemeingültiger UTC-Zeitstempelformatierung:


```
{
  "timestamp": "2024-09-23T18:59:44 CET",
  "machine": "Mjolnir",
  "state": "inuse local_lab_admin"
```

```
}
{
  "timestamp": "2024-10-17T20:19:44 CET",
  "machine": "Mjolnir",
  "state": "free"
}
{
  "timestamp": "2024-10-17T20:19:46 CET",
  "machine": "Mjolnir",
  "state": "inuse local_lab_admin"
}
```

Mit lokaler Zeitzone

Oder zum Beispiel mit der Zeitzone `Europe/Berlin`:

```
TZ=Europe/Berlin jq '.|.timestamp |= strflocaltime("%Y-%m-%dT%H:%M:%S %Z")' /var/log/bffh-audit.json
```

```
{
  "timestamp": "2024-09-23T18:59:44 CET",
  "machine": "Mjolnir",
  "state": "inuse local_lab_admin"
}
{
  "timestamp": "2024-10-17T20:19:44 CET",
  "machine": "Mjolnir",
  "state": "free"
}
{
  "timestamp": "2024-10-17T20:19:46 CET",
  "machine": "Mjolnir",
  "state": "inuse local_lab_admin"
}
```

oder noch lesbarer durch Umstellen des Formats:

```
TZ=Europe/Berlin jq '.|.timestamp |= strflocaltime("%d.%m.%Y %H:%M:%S Uhr")' /var/log/bffh-audit.json
```

```
{
  "timestamp": "08.12.2024 00:57:27 Uhr",
  "machine": "zam-raum1-ecke8-macgyver",
  "state": "free"
}
{
  "timestamp": "08.12.2024 00:57:29 Uhr",
  "machine": "zam-raum1-ecke8-macgyver",
  "state": "inuse Admin"
}
{
  "timestamp": "08.12.2024 00:57:30 Uhr",
  "machine": "zam-raum1-ecke8-macgyver",
  "state": "free"
}
```


> Zum Parsen und Patterns testen empfehlen wir das Online-Tool https://jqplay.org

Grafische Darstellung des Audits und Auswertung

Für die Dashboard-Anzeige in Grafana können die Werkzeuge Alloy und Loki verwendet werden.

Achtung: Das Parsen des Audit Logs ist nur so lange zuverlässig, wie das Log nicht gelöscht oder rotiert wird. Für eine Langzeitarchivierung und eine tiefergehende Analyse von Statistiken sollte u.U. in Betracht gezogen werden den Audit in einer Datenbank zu speichern, z.B. PostgreSQL oder MariaDB. Mit Hilfe einer Datenbank könnten dann tiefergreifende Auswertungsabfragen programmiert werden, die sich dann wiederrum z.B. in Grafana grafisch auswerten und exportieren ließen, oder aber für die Abrechnung genutzt werden könnten. Es sollte auch beachtet werden, dass immer größer werdende json-Dateien sich nicht mehr performant parsen lassen, um darüber akkumulierte Gesamtstatistiken zu erzeugen.

Statistiken aus dem Audit gewinnen

Statistikabfragen könnten z.B. sein:

- Wer nutzt den Space bzw. bestimmte Ressourcen am meisten oder wenigsten?
- Welche Ressourcen werden generell am häufigsten genutzt?

- Welche Ressourcen werden generell am längsten genutzt?
- wer nutzt am meisten Dinge im Space gleichzeitig?
- Wann ist der Space in Nutzung und wann ruht er?
- wieviele Maschinenstunden kommen pro Tag, Woche, Monat, Jahr zusammen?

Log Rotation

Sind Log Files zu groß, werden sie in der Regel "rotiert" und in Archiven komprimiert, um Speicher zu sparen. Folgende Konfiguration kann angewendet werden. Wir heben maximal 10 als *.gz komprimierte Log-Archive auf. Die Logs werden automatisch rotiert, wenn sie 1 Megabyte überschreiten.

```
sudo vim /etc/logrotate.d/bffhd
```

```
/var/log/bffh/audit.json
{
	rotate 10
	size 1M
	copytruncate
	missingok
	notifempty
	compress
}
```

Logs manuell rotieren (testen)

```
sudo logrotate /etc/logrotate.d/bffhd -f
```

Weitere Links

Siehe auch Aktor: Logger (CSVlog)

Ausleihen and Transfer

Um die soziale Interaktion im Space zu fördern, ermöglicht FabAccess das Weitergeben von Claims. Benutzer können somit Ressourcen direkt an andere Benutzer weitergeben oder diese zum Zwecke der Ausbildung an andere Benutzer verleihen (englisch: "lending").

Diese Funktionen ermöglichen eine flexible Nutzung der Ressourcen und fördern die Zusammenarbeit zwischen den Benutzern. Zum Beispiel kann ein Benutzer, der eine Ressource nicht mehr benötigt, diese einem anderen Benutzer übertragen, der sie gerade benötigt, oder ein Ausbilder kann einem Auszubildenden vorübergehend Zugriff auf eine Ressource gewähren, um bestimmte Fähigkeiten zu erlernen.

Durch diese Funktion kann die soziale Interaktion über FabAccess nachverfolgt und unterstützt werden.

Auswahl Programmiersprache und Framework für BFFH

Dieses Dokument basiert auf dem ADR-Commit https://gitlab.com/fabinfra/fabaccess/bffh/-/commit/a62a5678dc35cb32c899483cfab48ce218694d11 und wurde ins Deutsche übersetzt.

Der technische Hintergrund: Entscheidung über die zu verwendende Programmiersprache und das Framework für BFFH / den Backend-Code von FabAccess

Kontext und Problemstellung

Programmiersprachen-Diskussionen sind das perfekte Thema für einen Fahrradausflug. Unabhängig davon muss irgendwann eine Entscheidung getroffen werden, damit die Leute anfangen können, Code zu schreiben und allgemein nützlich zu sein. Da FabAccess als Projekt mit mehreren potenziellen Entwicklern und noch mehr Beteiligten, die am Erfolg der Software interessiert sind, gestartet wurde, war diese Diskussion besonders pikant. Die entsprechenden Diskussionen fanden im Zeitraum von etwa November 2019 bis März 2020 statt.

Entscheidungstreiber

- Verfügbare Entwickler, sowohl kurz- als auch mittelfristig, sollte das Projekt ein erfolgreiches Standbein für deutsche/europäische/irdische Makerspaces werden
- Einstiegshürde für neue/zusätzliche Entwickler.
- Tooling-Unterstützung, insbesondere für Nicht-Entwickler (Dokumentation, einfache Selbstkompilierung, …)
- Bibliotheksunterstützung / Ökosystem in der Problemdomäne
- Bestehende Projekte, die erweitert werden könnten, um das Rad nicht noch einmal zu erfinden
- Sprachliche Merkmale

Geprüfte Optionen

- Python
- TypeScript
- Rust
- Erlang

- Ruby
- Haskell

Entscheidungsergebnis

Gewählte Option: „Rust“, weil das Projekt schließlich größtenteils von FabInfra entwickelt wird, dessen Kosten-Nutzen-Analyse der Typensicherheit Vorrang vor der Entwicklungsgeschwindigkeit einräumt.

Positive Konsequenzen

- Die Typsicherheit von Rust ist im Vergleich zu allen anderen Optionen mit Ausnahme von Haskell sehr hoch
- Eine einzige statisch gelinkte Binärdatei mit sehr wenigen Abhängigkeiten macht die Bereitstellung trivial
- Kompilierte Software nahe an Baremetal ermöglicht den Betrieb des Servers auf leistungsschwacher Hardware wie Raspberry Pis, selbst bei mittelgroßen Einsätzen.

Negative Konsequenzen

- Die Geschwindigkeit der Entwicklung leidet kurzfristig. Rust ist eine Sprache mit niedrigerem Niveau als alle anderen evaluierten Optionen und erfordert mehr Denkarbeit und Entwicklerzeit, um Funktionen auf hohem Niveau zu implementieren.
- Der verfügbare Entwicklerpool ist klein und teuer. Rust ist keine gute Einstiegssprache und für Entwickler, die keine Erfahrung mit C++ haben, vergleichsweise schwer zu erlernen. Es können zwar Entwickler eingestellt werden, aber die Gehaltsvorstellungen von Rust-Entwicklern sind im Vergleich zu Python/Java hoch.
- Kleines Ökosystem / wenige Bibliotheken. Rust ist eine sehr junge Programmiersprache und verfügt, obwohl sie seit mehreren Jahren als die beliebteste Sprache auf stackoverflow.com eingestuft wird und kommerziell genutzt wird, über weniger Bibliotheken als die meisten anderen Optionen. Die Unterstützung für z.B. LDAP ist nicht ausgereift. Aber die Problemdomäne von FabAccess liegt in dem Bereich, für den Rust die meiste Unterstützung hat.

Vor- und Nachteile der Optionen

Python

- Gut, weil großer verfügbarer Entwicklerpool
- Gut, weil ein großer Teil der an der Entwicklung interessierten Entwickler Erfahrung mit Python hat
- Gut, weil Python leicht zu erlernen ist und eine gute Einstiegssprache darstellt
- Gut, weil Python über umfangreiche Bibliotheksunterstützung und ein auf Automatisierung ausgerichtetes Ökosystem verfügt
- Schlecht, weil vergleichsweise langsam. Langsamste oder zweitlangsamste Option in Betracht.
- Schlecht, weil die hochdynamische Typisierung und die mittelmäßige statische Analyse dazu führen können, dass Edgecase-Fehler lange Zeit verborgen bleiben
- Schlecht, weil Python wenig Kontrolle über den Absturzprozess bietet. Das Abfangen von Exceptions ist sehr grob.

TypeScript

- Gut, weil stärkere Typisierung als bei Python und Ruby
- Schlecht, weil interpretierte Sprache und zwar schneller als Ruby/Python, aber immer noch viel langsamer als die kompilierten Optionen.
- Schlecht, weil das Node.js Ökosystem groß ist, aber nicht so sehr auf Automatisierung auf niedriger Ebene ausgerichtet ist.

Rust

- Gut, weil Rust ein gutes Typensystem hat, das es erlaubt, viele Fehler und Abstürze durch die in den Compiler eingebaute statische Analyse zu verhindern.
- Gut, weil Rust-Code sehr effizient ist.
- Gut, weil viele Leute sehr daran interessiert sind, Rust zu lernen und ihnen nur ein Grund dafür fehlt.
- Schlecht, weil die Geschwindigkeit der Entwicklung langsamer ist als bei den meisten anderen Optionen.
- Schlecht, weil nur ein Entwickler in der Gruppe Erfahrung mit Rust hat.
- Schlecht, weil der verfügbare Entwicklerpool klein ist.

Erlang

- Gut, weil speziell für hochverfügbare, absturzsichere Software entwickelt
- Gut, weil es ein extrem feinkörniges Absturzsystem mit gekapselten Prozessen bietet, die sich nicht gegenseitig ausschalten können
- Gut, weil das Ökosystem und die Bibliotheken von Erlang generell auf die Problemdomäne von FabAccess ausgerichtet sind
- Gut, weil Erlang im Vergleich zu TypeScript, Pyton und Ruby eine umfangreiche statische Analyse ermöglicht

- Schlecht, weil Erlang ein schlechteres Typsystem hat als Rust und Haskell.
- Schlecht, weil nur ein Entwickler in der Gruppe Erfahrung mit Erlang hat.
- Schlecht, weil der Entwicklerpool der zweitkleinste ist.

Haskell

- Gut, weil das stärkste Typensystem viele Fehler und Abstürze durch die im Compiler integrierte statische Analyse verhindern kann
- Schlecht, weil nur ein Entwickler Erfahrung mit Haskell hat.
- Schlecht, weil der Entwicklerpool am kleinsten ist.

Links

- Die Diskussionen sind dokumentiert in:
  - Protokoll Jit.si-Konferenz 09.12.19
  - Pad "Roseguarden"
  - Pad BF²H "Grundlegendes"
  - JitSi Call 18.02

Cache (Zwischenspeicher)

In FabAccess gibt es eine Unterscheidung zwischen statischen und dynamischen Daten, die von Ressourcen besessen werden.

Dynamische Daten umfassen Zustände oder Messwerte, die von Ressourcen erzeugt werden.

Statische Daten hingegen sind Eigenschaften, die sich nicht regelmäßig ändern, sondern nur in größeren zeitlichen Abständen. Diese statischen Daten werden in FabAccess zwischengespeichert, um die Antwortzeiten des Servers zu reduzieren und Ressourcen in Clients schneller anzeigen zu können.

> **Hinweis:** Das Konzept zu Cache exitiert zwar, jedoch gibt es noch keine Spezifikation!

Claims, Notify und Interest (das Konzept vom "Anspruch erheben")

Das Konzept von Claims dient als Abstraktion des Verleihens einer Ressource in FabAccess. Ihr Hauptzweck besteht darin, die Möglichkeiten zu verwalten, wie Benutzer Zugriff auf eine Ressource erhalten und diesen Zugriff untereinander teilen können bzw. wie Maschinen voneinander abhängig gemacht werden. Sie vereinfachen den Ausleihprozess für den Space und dessen Nutzer. Claims enthalten sog. "Interests" und "Notifies".

Zum Beispiel kann konfiguriert werden, dass eine Bandsäge nur angeschalten werden kann, wenn auch die Absaugung an ist oder der Laserschneider nur einschaltbar ist, wenn die Kühlung aktiv ist.

Claim (Anspruch erheben)

Ein "Claim" in FabAccess repräsentiert den gewährten Zugriff auf eine Ressource.

Die Flexibilität von Claims ermöglicht es, verschiedene Szenarien des Ressourcenzugriffs effektiv zu unterstützen. Zum Beispiel können mehrere Benutzer gleichzeitig einen Claim für eine Ressource erhalten, was besonders in Umgebungen mit kollaborativem Arbeiten von Vorteil ist. Darüber hinaus bietet die Möglichkeit, sich in eine Warteschlange mit einem Interest einzutragen, eine praktische Lösung für Situationen, in denen die Verfügbarkeit einer Ressource begrenzt ist und Benutzer darauf warten müssen, darauf zugreifen zu können.

in weiterer wichtiger Aspekt von Claims ist ihre Fähigkeit, Ressourcen zwischen Benutzern zu transferieren oder auszuleihen. Diese Funktionen ermöglichen eine flexible Nutzung der Ressourcen und fördern die Zusammenarbeit zwischen den Benutzern. Zum Beispiel kann ein Benutzer, der eine Ressource nicht mehr benötigt, diese einem anderen Benutzer übertragen, der sie gerade benötigt, oder ein Ausbilder kann einem Auszubildenden vorübergehend Zugriff auf eine Ressource gewähren, um bestimmte Fähigkeiten zu erlernen.

Interest (Interesse anmelden)

Ein wichtiger Bestandteil des Claims-Konzepts sind "Interests", die Reservierungen abbilden, die entweder zeitbasiert oder auf einer Warteschlange basieren können. Diese Interessen bieten den Benutzern die Möglichkeit, einen zukünftigen Zugriff auf eine Ressource zu sichern, entweder basierend auf einer vordefinierten Zeit oder in der

Reihenfolge des Eintritts in die Warteschlange.

Mit einem Interest signalisiert der Nutzer dem System sein Interesse an einer bestimmten Ressource. Bei der nächsten Gelegenheit erhält der Nutzer entweder einen Claim auf die Ressource oder hat die Möglichkeit, seinen Interest auf einen Claim zu aktualisieren. Diese Flexibilität ermöglicht es den Benutzern, ihre Bedürfnisse anzupassen und auf Änderungen in der Verfügbarkeit der Ressourcen zu reagieren.

Notify (Benachrichtigen)

Das letzte Element des Claims-Konzepts ist der "Notify". Durch den Notify können Nutzer den Status einer Ressource einsehen und sich über Änderungen benachrichtigen lassen.

Der Notify ermöglicht es Benutzern, den aktuellen Zustand einer Ressource abzurufen und bei Bedarf auf Änderungen zu reagieren. Dies bietet eine wichtige Möglichkeit, den Zustand von Ressourcen zu überwachen und zeitnah auf relevante Ereignisse zu reagieren.

Durch die Möglichkeit, sich für Benachrichtigungen über Zustandsänderungen zu registrieren, können Benutzer effektiv mit den Ressourcen interagieren und sicherstellen, dass sie stets über wichtige Entwicklungen informiert sind.

Datenbank-Konzept (LMDB)

FabAccess BFFH nutzt derzeit eine auf Datenbank auf Dateibasis. Hierzu bedient es sich der Rust-Implementierung *lmdb_rs* für die original in C geschriebene LMDB (Lightning Memory-Mapped Database). Das ist eine eingebettete transaktionale Datenbank in Form eines Key-Value-Speichers.

BFFH schreibt in die Datenbank zwei wesentliche Informationsspeicher:

- `users`: Benutzerdatenbank (Nutzernamen, Passwörter, Cardkeys, Rollenzuweisungen)
- `states`: Ressourcenzustände

> **Wichtig zu wissen:** 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.

Federation (Föderation)

FabAccess ist als selbstständiges selbstgehostetes System für jeden Space gedacht, so können die Spaces selber über das komplette System verfügen und es eigenständig bis in kleinste Detail konfigurieren. Um die Zusammenarbeit von Spaces zu ermöglichen und zu fördern, können die einzelnen FabAccess Instanzen (BFFH) mit einander föderieren. Dadurch können Nutzer zwischen diesen Spaces mit verringertem Verwaltungsaufwand hin und her wechseln. Wir wollen so die Spezialisierung von Spaces ermöglichen, ohne dass Nutzer die Vorteile eines vollausgestatten Spaces verlieren. Auch kleineren Spaces wird so der Einstieg vereinfacht.

Föderation ist dabei als aktive Zusammenarbeit erdacht, also müssen sich die Betreiber dieser Spaces dazu entscheiden zu föderieren. Jeder Space kann so den Partner für die Föderation selber wählen.

Der FabAccess Client Borepin ist bereits insofern für Föderation ausgelegt, als dass er mehrere Serveradressen und Benutzer speichern kann. So ist ein Wechsel zwischen verschiedenen Spaces und/oder Nutzermodellen praktisch möglich.

Vorteile und Ziele der Föderation

- beliebiger Wechsel der Nutzer verschiedener Spaces trotz technisch getrennter FabAccess-Instanzen
- Zusammenarbeit von Spaces unterstützen - durch Teilen von Berechtigungen ihrer Nutzerschaften
- FabAccess verbreiten, ähnlich wie das Matrix Chat-Protokoll dies mit Kommunikation macht
- kann langfristig dabei helfen, Maschinendokumentationen und Maschineneinweisungen (Maschinenführerscheinen) zu vereinheitlichen
- Spart NFC Karten (FabFireCards) ein, denn die Karten können dann in alle föderierten Spaces genutzt werden
- Mit Föderation is theoretisch auch das Bilden von standortübergreifenden Clustern möglich, also eine Art Filialbildung (das ist ein wertungsfreier Fakt)
- ist dabei als aktive Zusammenarbeit erdacht: die Betreiber der Spaces können sich selbst entscheiden, ob sie föderieren wollen. Föderation ist also immer optional und kann zudem jederzeit wieder deaktiviert werden

Modi oder Stufen der Föderation

Die folgenden Stufen bauen aufeinander auf, so können sich Betreiber einfacher an eine Föderation wagen und die Vorteile mit jeder Stufe erfahren.

1. Stufe (Authentifizierung: Teilen von Nutzern)

- Nutzer aus anderen Spaces können ihre FabFireCard auch in deinem Space verwenden
- Die Nutzer sind auch in deinem Space registriert
- Der föderierte Space übernimmt die Authentifizierung für den anderen Nutzer (wie bei SSO)

Unter dem Teilen von Nutzern verstehen wir, dass Nutzer von Space B bei dir im Space A sich mit der FabFireCard aus Space B authentifizieren können. Um bei dir im Space A die Maschinen nutzen zu dürfen, müssen die Nutzer aus Space B sich bei dir anmelden und deinen Nutzungsvertrag ausfüllen. Da die Nutzer aus Space B FabAccess schon kennen, werden Sie sich selbstständig registrieren und du musst nur nach einer kurzen Prüfung der Daten diese Nutzer akzeptieren. Dabei erhältst du immer die Kontaktdaten deiner Nutzer und bist nicht auf die Nutzerdaten des anderen Spaces angewiesen, sollte bei dir mal ein Unfall passieren. Die Authentifizierung der Nutzer übernimmt dann die BFFH Instanz des Space B. Um nicht immer mit jedem Space sofort aktiv föderieren zu müssen, ermöglichen wir es dir auch das Teilen von Nutzern für unbekannte Spaces zu erlauben. So kannst du nach einer aktiven Föderation mit dem anderen Space gleich die Nutzerdaten übernehmen, um so nicht noch mal alle Nutzer anzupassen. Bei einer Föderation sind nämlich die IDs der Nutzer in den beiden Spaces gleich und ermöglichen den nächsten Schritt in der Föderation.

2. Stufe (1. Stufe ergänzt um Teilen von Berechtigungen)

- Nutzer aus anderen Spaces können ihre FabFireCard und ihre Gruppen auch in deinem Space verwenden
- Die Nutzer sind auch in deinem Space registriert
- die Gruppen des anderen Space können in deinem Space verwendet werden, wenn zum Beispiel die gleichen Maschinen im Besitz sind. Das spart Zeit bei der Einweisung und der Dokumentation.

Nach dem das Teilen von Nutzern erfolgt ist und du sich mit dem Betreiber des Space B gut verstehst, stellt ihr beide fest, dass ihr einige gleiche Maschinen in euren Spaces besitzt. Da du dem Betreiber von Space B auch bei den Einweisungen von den Maschinen vertraust, könnt ihr in der 2. Stufe auch Berechtigungen teilen. Somit müssen Nutzer von Space B, die bei dir die gleiche Maschine wie im Space B verwenden wollen, von dir nicht noch mal eingewiesen werden. Das spart Zeit bei dir und bei den Nutzern von Space B. Du behältst dabei die Kontrolle, welche Gruppen von Space B bei dir im Space A welche

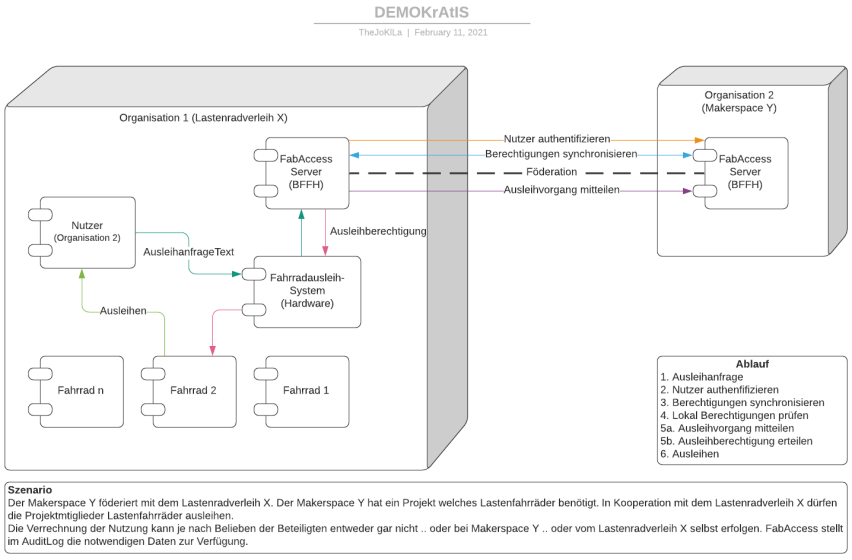
Maschinen nutzen können, somit kannst du dir sicher sein, dass die Nutzer wirklich eingewiesen sind.

3. Stufe (2. Stufe ergänzt um Teilen von Abrechnungen)

- Nutzer aus anderen Spaces können ihre Karte und ihre Gruppen auch in deinem Space verwenden
- Die Abrechung für diese Nutzter übernimmt dabei der andere Space für deinen Space. Über beispielsweise monatlich gesammelte Differenzrechnungen kann gegen den andere Spacecs abgerechnet werden. Das spart Zeit bei der Buchführung und die Nutzer können auch Ihre Rechnungen besser im Blick behalten.

Das Teilen von Abrechnungen stellt die letzte Stufe der Föderierbarkeit da. Da wir uns bei dem Thema noch nicht vollständig sicher sind, inwieweit das umgesetzt werden kann, steht es noch in Klammern. Die Idee zu der Föderation ist, dass Nutzer von Space B, die bei dir eine kostenpflichtige Maschine nutzen, ihre Abrechnung von Space B erhalten und nicht von dir. Das Geld der Nutzer von Space B kannst du dann mit dem Betreiber von Space B zum Beispiel monatlich verrechnen. Der Vorteil für den Nutzer liegt dabei darin, dass sie nicht bei jedem Space ihre Abrechnung machen müssen, sondern gesammelt bezahlen können.

Entwurf für einen Lastenfahrradverleih - Ablaufbeispiel "DEMOKrAtIS"



Wie sicher ist Föderation?

Föderation benötigt Vertrauen, jedoch möchten wir nicht, dass dieses Vertrauen einfach missbraucht werden kann. Daher ist der oben beschriebene Ansatz, dass du jeden deiner Nutzer auch kennst.

Sicherheit des Servers

Für den Austausch von Benutzern und Berechtigungen ist ein sicherer Datenaustausch über das Internet notwendig. Das bedeutet auch, dass ein Dienst Ziel eines Angriffs von außen sein kann. Sobald der Dienst auf einem geöffneten Port lauscht, muss sichergestellt sein, dass nur authorisierte Server bzw. Personen darauf Zugriff haben und keinen Missbrauch begehen. Daraus ergeben sich verschiedene Fragen und Anforderungen, wie zum Beispiel

- erfolgt der Informationsaustausch zwischen den föderierenden Servern auf dem gleichen Port oder gibt es hierfür z.B. eine https-verfügbare API (z.B. REST, Json, XML RPC, ...)?
- wie wird das Verhalten gemonitored und Schadverhalten abgewehrt (z.B. Greylisting, Blacklisting, DDoS Vermeidung, Bruteforce-Login Attacken, Fail2Ban Jails, Firewall-Regeln...)
- welche APIs sind unterinander kompatibel? Nicht jeder Space wird schnell genug seinen Server auf die aktuelle Version updaten können
- Wie isolieren wir den FabAccess-Server ausreichend gegen Angriffe von außerhalb? Immerhin hat bffh Zugriff auf physische Geräte durch das Schalten von Strom an Aktoren

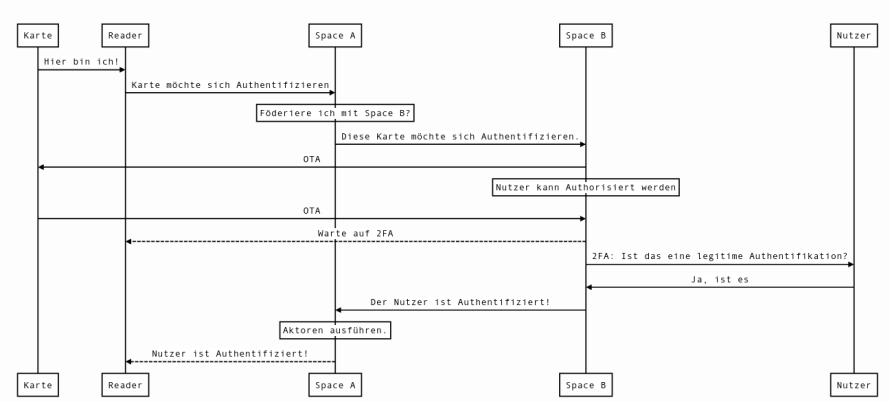
Sicherheit von FabFire Karten

Um die Authentifizierung sicher zu gestalten, haben wir uns für die etwas teureren NXP MIFARE DESFire EV2 Karten entschieden (ca. 1-5 € pro Karte bzw. Tag). Diese haben den Vorteil, dass sie ein Ende-zu-Ende (E2EE) Verfahren unterstützen und bisher als ungehacked gelten. Somit musst Du nicht die Schlüssel deiner Karten an jeden Reader in einem anderen Space verteilen, sondern deine BFFH Instanz kommuniziert verschlüsselt direkt mit der Karte. Auf den FabFireCards befinden sich auch keine Informationen der Nutzer, sondern nur der DNS-Eintrag deines Space sowie eine pseudonymisierte Nutzer ID. Somit wollen wir die technische Seite der Föderation absichern und eine Zusammenarbeit möglichst einfach gestalten.

Die Anforderungen an ein föderierbar einsetzbares SmartCard-System sind hoch, da so die jeweils andere Partei keine einfache Validierung der Karte durchführen kann. Auch das Authentifizieren mit einem Schlüssel gestaltet sich über Föderationen hinweg schwierig, da der Reader dem einen Space gehört und die Karte dem anderen. Ein Reader hat daher

nicht immer die Schlüssel für die gelesenen Karten. Die DESFire-Karte besitzt ein Feature, mit dem eine Karte via OTA-Verfahren (Over-the-Air) direkt mit einem entfernten Server kommunizieren kann, ohne dass der Reader Schlüssel für die Karte benötigt. Der Reader übernimmt dabei nur die Rolle eines Proxys (Brücke/Vermittler). Mit diesem Ansatz stellen wir ein sicher föderierbares Kartensystem, was wir so bisher noch nicht im OpenSource-Bereich finden konnten.

Das föderative Verfahren für FabFire Karten wurde in folgendem Grobschema veröffentlicht:



Fragen und Ansprüche an Föderation und potentielle Probleme

- muss stabil sein
- es muss sicher sein (siehe oben)
- Benutzer und Berechtigungen anderer Spaces können sich ändern. Nutzer könnten dem Space nicht mehr angehören. Wie halten wir den Informationsstamm ausreichend aktuell?
- es müssen verschiedene Server-/API-Versionen interoperabel funktionieren
- es braucht eine Whitelist bzw. ggf. sogar Blacklist für alle Spaces, die für FabAccess nutzbar sein sollen
- Vertrags-, Datenschutz- und Haftungsfragen: ist die Übertragung der Nutzerdaten an andere Spaces geregelt, und wenn ja, wie? Welche vertraglichen Regeln müssen für eine Föderation mit FabAccess festgehalten werden?
- Föderierte Spaces arbeiten ggf. sehr unterschiedlich, was Maschineneinweisungen und Co. angeht - wie kann sichergestellt werden, dass dazu übergreifende Standards geteilt werden?

Stand der Föderation

>

> Das Feature Föderation ist aktuell (Stand Dezember 2024) noch nicht in der beschriebenen Art nutzbar in FabAccess implementiert. Ein direkter Datenaustausch zwischen mehreren BFFH-Servern findet nicht statt und es gibt auch kein zentrales Server-Register.
>
> Möglich ist bereits im Borepin Client das Speichern von verschiedenen Verbindungsprofilen (Serveradressen + Kombination Nutzer/Passwort) unterschiedlicher Spaces.

Wer nutzt heute schon FabAccess?

Potentielle Nutzer für FabAccess-Föderation findest du unter Mitmachen / Unterstützen / Join the Community

Measurements (Messwerte)

Durch "Measurements" in FabAccess werden Daten von Ressourcen gesammelt, um deren Leistung und Nutzung zu erfassen. Das Ziel ist es, dass FabAccess diese Messwerte sammelt und an die Nutzer weitergibt.

Die Funktion Measurements ermöglicht es FabAccess, wichtige Daten über die Nutzung von Ressourcen zu erfassen und den Nutzern zur Verfügung zu stellen. Dies umfasst Informationen wie Betriebsstunden, Auslastung und andere Leistungsindikatoren, die es den Nutzern ermöglichen, die Effizienz und Produktivität ihrer Arbeitsprozesse zu optimieren.

> **Hinweis:** Das Konzept zu Measurements exitiert zwar, jedoch gibt es noch keine Spezifikation!

RBAC (Benutzerrollen und Berechtigungen)

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, ufassend und flexibel abbilden.

Das Berechtigungssystem ermöglicht es, die Zugriffe auf Ressourcen nur für bestimmt Nutzer zu erlauben. Somit kann darüber die Einweisungen für gefährliche Ressourcen abgebildet werden. Auch die Abbildungen von einfachen Ressourcen, die keine Einweisung benötigen, ist möglich.

Berechtigungsübersicht

Die Resourcen haben folgende Berechtigungen, die zugewiesen werden und dem Nutzer folgendes ermöglichen:

disclose (offenlegen)
die Resource über die API zu entdecken, also in einer Liste zu erhalten. Mit dieser Berechtigung können Nutzer die Ressource in der Übersichtsliste der App sehen. Ohne diese Berechtigung wird die Ressource in der Liste nicht angezeigt.

read (lesen)
statische Informationen aus der Resource erhalten**.** Diese Berechtigung ermöglicht es Nutzern, Informationen zur Ressource abzurufen. Dazu gehört das Auslesen des aktuellen Zustands der Ressource und das Lesen weiterer Informationen wie Wiki-Links. Es ist wichtig zu beachten, dass ein Nutzer ohne disclose, aber mit read, die Ressource dennoch auslesen kann. Diese Funktion kann z.B. durch QR-Codes auf der Ressource als Präsenzfunktion genutzt werden.

write (schreiben)
Die write-Berechtigung ermöglicht es einem Nutzer, die Ressource zu verwenden. Auch das Ändern von Zuständen der Ressource wird dadurch möglich.

manage (verwalten)
einen claim zu überschreiben. Mit dieser Berechtigung kann die Ressource verwaltet werden. Ein Nutzer mit dieser Berechtigung kann eine verwendete Ressource freigeben oder als defekt markieren.

Nicht implementierte Berechtigungen

Folgende Berechtigungen sind konzipiert, aber aktuell nicht implementiert:

notify (benachrichtigen)
Notify aufrufen, um statische Informationen zu erhalten

claim (Anspruch erheben)
Die Resource leihen und Traits auszuführen

transfer (transferieren)
Die Resource an einen anderen Nutzer weitergeben

lend (ausleihen)
Die Resource an andere verleihen, die nicht die Berechtigung haben die Resource zu verwenden

Plugins

Die modulare Struktur von FabAccess benötigt Plugins, um das System an die Bedürfnisse des Spaces anzupassen. Plugins ermöglichen die Erweiterung der Grundfunktionen von FabAccess und die Integration zusätzlicher Features, ohne dass die Kompatibilität des Kernsystems beeinträchtigt wird.

Plugins können an folgenden Hauptstellen eingesetzt werden:

- Aktoren und Initiatoren
- Audit Log (Revisionsprotokoll)

Projekte

"Projects" in FabAccess sollen die Zusammenarbeit zwischen den Nutzern fördern und gleichzeitig die Abrechnung von Ressourcennutzungszeiten verbessern. Ein wichtiger Aspekt der Projektstruktur ist die Möglichkeit für Nutzer, Claims innerhalb desselben Projekts miteinander zu teilen, um gleichzeitig auf Ressourcen zugreifen zu können.

Durch die Zuweisung von Nutzern zu Projekten können Teams effizient zusammenarbeiten und ihre Ressourcen optimal nutzen. Das Teilen von Claims innerhalb eines Projekts ermöglicht es den Teammitgliedern, nahtlos auf benötigte Ressourcen zuzugreifen und gemeinsam an Projekten zu arbeiten. Darüber hinaus erleichtert diese Funktionalität die Abrechnung von Ressourcennutzungszeiten, da die Nutzung der Ressourcen innerhalb eines Projekts besser nachverfolgt und zugeordnet werden kann.

> **Hinweis:** Das Konzept zu Projekten exitiert zwar, jedoch gibt es noch keine Spezifikation!

Terminals

Terminals in FabAccess bieten einen eingeschränkten Zugang zum Server. Diese Terminals können nur auf die ihnen zugewiesenen Ressourcen zugreifen und haben die Möglichkeit, Ressourcen an andere Benutzer auszuleihen.

Aufgrund ihrer eingeschränkten Zugriffsrechte sind Terminals ideal für die Authentifizierung in FabFire. Durch die Verwendung von Terminals als Authentifizierungsmethode können Benutzer sicherstellen, dass nur autorisierte Personen auf die Ressourcen zugreifen und diese nutzen.

Die Verwendung von Terminals zur Authentifizierung bietet eine zusätzliche Sicherheitsebene und hilft, die Integrität des Systems zu wahren. Darüber hinaus ermöglicht es eine effiziente Verwaltung der Ressourcennutzung, indem der Zugriff nur autorisierten Benutzern gewährt wird.

Unsere eigene Terminal-Implementierung nennt sich FabReader und verwendet dafür Mifare DESFire EV2 Karten.

Externe Authentifikation

Das Authentifizieren in FabAccess basiert vollständig auf SASL und unterstützt daher verschiedene Mechanismen.

Die Rollen, die aus den Gruppen bei LDAP und OAuth abgeleitet werden, werden additiv zu denen betrachtet, die intern in FabAccess vergeben werden.

Authentifikation per KeyCloak

Das folgende Tool nutzt Keycloak im Zusammenhang mit FabReadern und umgeht die eigentlich FabAccess-Funktionalität. Ggf. hilft dieses Projekt jedoch weiter bei der eigenen Systemintegration: https://gitlab.com/sfz.aalen/infra/fabaccess

Authentifikation per OAuth

> In Entwicklung seit 10/2024 - Ansprechpartner: Jonathan Krebs

Authentifikation per LDAP

> In Entwicklung seit 10/2024 - Ansprechpartner: Jonathan Krebs

Es besteht keine native LDAP Integration. Es gibt jedoch Tools zum Importieren von LDAP-Benutzern in die bffh-Datenbank. Siehe LDAP Anbindung.

Authentifikation per FabFire

FabFire ist eine Entwicklung, die auf NXP Mifare DESFire basiert und die Anwendung von Karten nutzt, um Benutzer über das OTA (Over the Air)-Verfahren zu authentifizieren.

Nutzerverwaltung

Die Nutzerverwaltung ermöglicht es, den Überblick über die Nutzer im Space nicht zu verlieren. FabAccess ist als SmartCard System gedacht, somit erhält jeder Nutzer eine NXP Mifare DESFire (FabFireCard) Karte, um sich an der Ressource zu authentifizieren.

Um den administrativen Aufwand zu verringern, können sich Nutzer selbstständig anmelden und können dann nach der Prüfung der Daten durch den Betreiber freigeschaltet werden. Die Schließung eines Nutzungsvertrages wird dabei abgebildet.

Mit der FabFireCard können Nutzer sich sowohl an den Ressourcen authentifizieren als auch an unserem Client Borepin.

Sensoren

Sensoren sind abstrahierte Datengeneratoren, die es BFFH ermöglichen, Ereignisse mit dem in der realen Welt auftretenden Verhalten zu verknüpfen und Entscheidungen darauf zu gründen. Ziel ist das regelmäßige Abfragen von Sensordaten, die dann für Folgeentscheidungen weiterverarbeitet werden können.

> Das Konzept der Sensoren ist aktuell nur als Draft in einem Ticket formuliert, aber bis auf einen leeren Unterodner im bffh-Projekt nicht implementiert

Zweck

Insbesondere bei Abhängigkeiten und ähnlichem ist es wichtig, dass sich Ressourcen in einem bestimmten (gemessenen) Zustand befinden und nicht nur von ein und derselben Person genutzt werden. Sensoren ermöglichen solche Muster, indem sie z.B. einer Pumpe erlauben, BFFH mitzuteilen, dass sie die Nennleistung erreicht hat oder dass sie damit aufgehört hat, oder einem 3D-Drucker, der BFFH mitteilt, dass der aktuelle Druckauftrag beendet ist.

Limitierungen

BFFH arbeitet nur mit abgetasteten Ereignissen und hat keine Vorstellung von absoluter oder gar quantifizierbarer Zeit. Ereignisse können in Relation zueinander geordnet werden, aber die Zeit zwischen zwei Ereignissen ist nicht bestimmbar. Daher macht es absolut keinen Sinn, das reaktive Ereignisnetz, das den Kern des BFFH bildet, für Histogramme oder ähnliche Messungen zu verwenden. Stattdessen sollten die Sensoren so weit wie möglich zusammengefasste Daten liefern, die sofort für die Entscheidungsfindung verwendet werden können. So ist z.B. „Kühlpumpe hat Nennleistung erreicht“ gut, „Kühlpumpe fördert 12l/min“, die jede Sekunde zurückkommt, nicht.

Zustände (Traits)

Traits bieten die Möglichkeit, den Zustand von Ressourcen zu ändern. Ressourcen können mehrere Traits besitzen und diese kombiniert nutzen. Mit Traits erhalten Nutzer Zugriff auf die Ressource, nachdem sie einen Claim erhalten haben. Dabei können Traits verwendet werden, um Ressourcen aus bestehenden Traits zusammenzusetzen oder spezifische Traits zu implementieren. Um eine optimale Anzeige der Traits für Nutzer in Clients zu ermöglichen, kann einer Ressource ein "Hint" hinzugefügt werden. Dieser ermöglicht es einem Client, eine verbesserte Ansicht der Ressource für Nutzer zu generieren.

Traits und OIDs

Traits werden anhand einer OID (Object Identifier) bereitgestellt. Die OID-Struktur von FabAccess folgt dem Schema:

```
1.3.6.1.4.1.61783.612.1.2
              |    |  | |
  RLKM UG PEN ┘    |  | |
                   |  | |
 FabAccess subtree ┘  | |
                      | |
               Traits ┘ |
                        |
               Doorable ┘
```

Für das Projekt wurde eine gesonderte Private Enterprise Numer (PEN) bei IANA auf "FabInfra" registiert. Das feste PEN-Präfix lautet `1.3.6.1.4.1`. Die PEN-Nummer für RLKM lautet `61783` (Siehe https://www.iana.org/assignments/enterprise-numbers/?q=61783.). Das Projekt FabAccess hat eine fest vergebene Unternummer `612`. Danach folgt die Unterklasse `1` (Traits) und dann die möglichen Zustandsnummern (Enumeration) `0` ... `5` (z.B. Doorable, Locatable, ...).

> In älteren Versionen der API wurde die PEN 48398 genutzt ("Paranoidlabs").

Die OID-Bezeichner werden in der Zustandsdatenbank abgespeichert. Siehe auch Datenbeispiel eines Dumps.

Übersicht über die vorhandenen Traits und OIDs

In FabAccess gibt es bereits vordefinierte Traits für grundlegende Funktionen, mit denen viele Zustände von Ressourcen abgebildet werden können.

FabAccess-API

> Aktuell ist diese API in Verwendung. Siehe
>
> https://gitlab.com/fabinfra/fabaccess/fabaccess-api

In diesem API-Modell gibt es nur die folgenden 7 Zustände (Traits) von 0 bis 6:

- 0: `Free` - eIne Ressource frei verfügbar für Nutzer mit Zugriff machen
- 1: `InUse:<UserId>` - eine Ressource durch einen Nutzer "In Benutzung" setzen
- 2: `ToCheck:<UserId>` - eine Ressource auf "muss überprüft werden" setzen (z.B. ob Reinigung nach Benutzung notwendig ist) - **Achtung**: Dieser Zustand kann nur über die API und nicht über Client genutzt werden (weil nicht implementiert. Siehe auch eine Ressource bedienen)
- 3: `Blocked:<UserId>` - eine Ressource blockieren (z.B. weil die Maschine defekt ist)
- 4: `Disabled` - eine Ressource deaktivieren (z.B., weil die Ressource zum aktuellen Zeitpunkt zu laut wäre und deshalb die Benutzung verboten ist)
- 5: `Reserved:<UserId>` - eine Ressource für die spätere Nutzung vorreservieren - **Achtung**: dieser Zustand kann aus dem Borepin Client nicht gesetzt werden (Siehe auch eine Ressource bedienen), jedoch im Process Aktor und im Python Process Actor Template schon
- 6: `totakeover` - eine Maschine von einem anderem Nutzer übernehmen - **Achtung** : dieser Zustand ist weder in BFFH, noch im Process Aktor, noch im Python Process Actor Template implementiert. Dieser Zustand existiert aktuell nur in der API-Beschreibung.

Neben den definierten Zuständen gibt es auch die Möglichkeit, direkte Rohdaten zu liefern:

- `Raw:<data>` - Ressourcen können bei Statuswechsel auch direkt mit Binärdaten versorgt werden, um zum Beispiel spezielle Operationen wie das Übersenden einer STL-Datei an einen 3D-Drucker zu erlauben

> Die API verwendet in einen OID-Wert (value) `1.3.6.1.4.1.48398.612.2.4` vom OID-Typ (type) `1.3.6.1.4.1.48398.612.2.14` für den Zustand (state) im Allgemeinen.

Prodable

Kommt vom englischen Wort "prodded" und bedeutet soviel wie "anstubsen". Ein Prodable ist also etwas Anstubsbares. Das lässt sich im Sinne einer Ereignisschleife verstehen, in der eine Sache die Möglichkeit hat, etwas zu tun. Wird im BFFH Server verwendet - es findet Nutzen beim FabLock. Dort hat es den Zweck vor dem Öffnen eines Schließfachs in einem Spind mit mehreren Fächern vorher kurz eine LED aufleuchten zu lassen, die signalisiert, wo genau sich das Fach befindet.

Prodable protokollieren keine Ressourcennutzung durch einen Nutzer.

Der `Prodable` Trait existiert namentlich so in der neuen C# API nicht mehr - er wurde in `Locatable` umbenannt. Weiterhin gibt es spezialisierte Traits `Doorable` für Eingangstüren und `Lockers` für Schließfächer.

FabAccess-API-cs

> Diese API Version ist aktuell noch nicht in Verwendung. Siehe https://gitlab.com/fabinfra/fabaccess/fabaccess-api-cs

Claimable (OID 1.3.6.1.4.1.61783.612.1.0)

Der Trait "Claimable" stellt einen Sonderfall dar, da er dazu dient, dass sich ein Nutzer über den Claim-Zustand einer Ressource informieren kann.

Nutzer können auf diesem Trait keine Aktionen ausführen, sondern lediglich den Zustand abfragen.

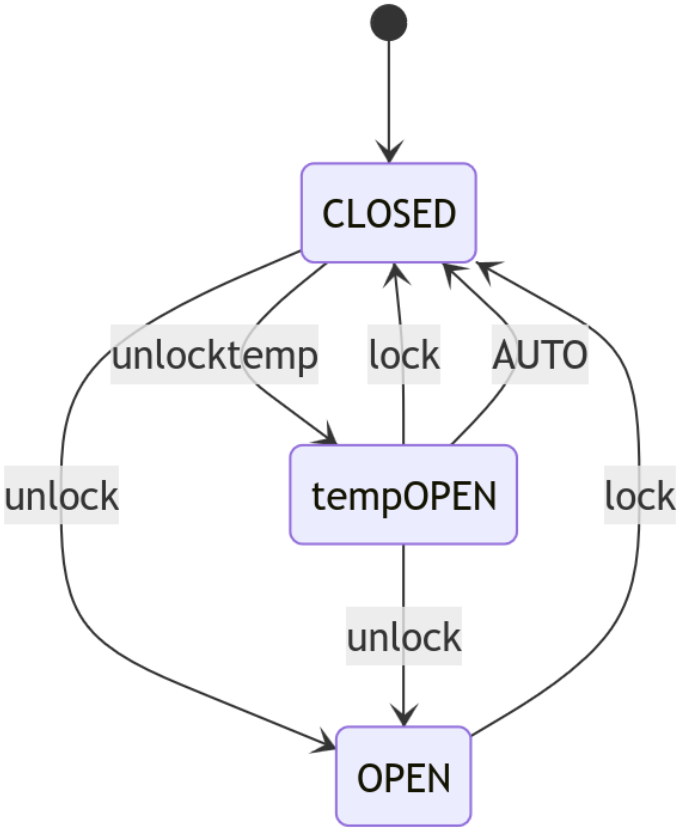
Powerable (OID 1.3.6.1.4.1.61783.612.1.1)

"Powerable" ist der grundlegendste Trait, den FabAccess unterstützt. Er dient dazu abzubilden, ob eine Ressource eingeschaltet bzw. mit Strom versorgt ist.

Doorable (OID 1.3.6.1.4.1.61783.612.1.2)

Der Trait "Doorable" ermöglicht die Abbildung von Türen oder anderen Schließsystemen. Dabei besteht die Möglichkeit, kurzzeitige Öffnungen zu realisieren. Die genaue Zeitdauer, für die die Ressource geöffnet wird, wird dabei vom Server bestimmt.

States



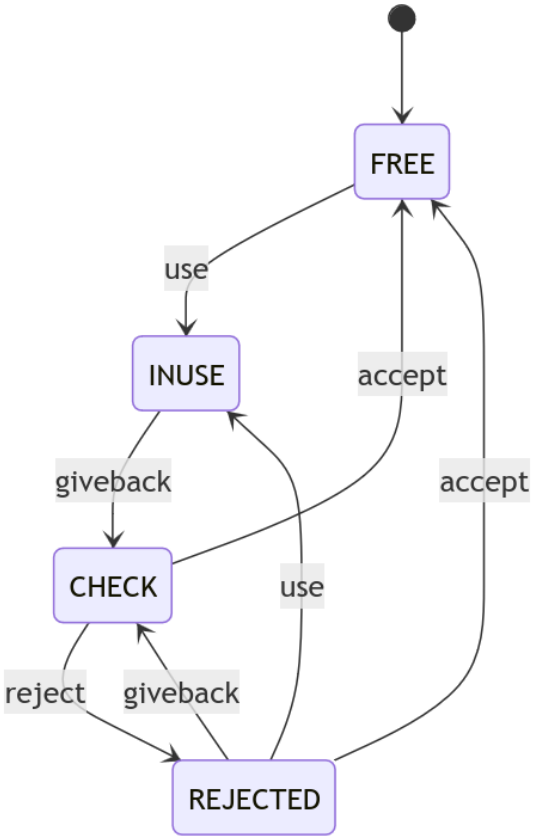
als Mermaid

```
stateDiagram
    [*] --> CLOSED
    CLOSED --> OPEN: unlock
    OPEN --> CLOSED: lock
    CLOSED --> tempOPEN: unlocktemp
    tempOPEN --> OPEN: unlock
    tempOPEN --> CLOSED: lock
    tempOPEN --> CLOSED: AUTO
```

Checkable (OID 1.3.6.1.4.1.61783.612.1.3)

Der komplexere Trait "Checkable" ermöglicht die Abbildung von Ressourcen, die nach der Benutzung überprüft werden müssen. Bei einer Überprüfung durch einen berechtigten Nutzer kann die Ressource entweder für alle wieder freigegeben oder zurückgewiesen werden. Falls die Ressource zurückgewiesen wurde, kann der ursprüngliche Nutzer die Ressource weiterhin verwenden oder erneut zur Überprüfung einreichen, nachdem die Fehler behoben wurden.

States



als Mermaid

```
stateDiagram
    [*] --> FREE
    FREE --> INUSE: use
    INUSE --> CHECK: giveback
    CHECK --> FREE: accept
    CHECK --> REJECTED: reject
    REJECTED --> INUSE: use
    REJECTED --> CHECK: giveback
```

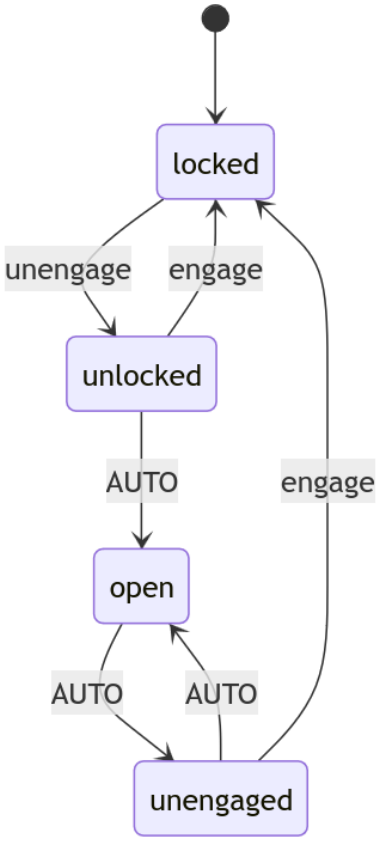
```
    REJECTED --> FREE: accept
```

Lockers (OID 1.3.6.1.4.1.61783.612.1.4)

"Lockers" ist einer der komplexeren Traits, mit dem Schlösser von Ressourcen genauer abgebildet werden können. Der Nutzer kann die Ressource nicht in jedem Zustand zurückgeben; erst wenn alles korrekt zurückgegeben wurde, kann die Ressource auch wieder gesperrt werden.

Die Zustandsänderungen zwischen den vom Nutzer verwendbaren Übergängen können von einem Initiator herbeigeführt werden.

States



als Mermaid

```
stateDiagram
    [*] --> locked
    locked --> unlocked: unengage
    unlocked --> locked: engage
```

```
unlocked --> open: AUTO
open --> unengaged: AUTO
unengaged --> locked: engage
unengaged --> open: AUTO
```

Locatable (OID 1.3.6.1.4.1.61783.612.1.5)

Der Trait "Locatable" ermöglicht die Identifizierung von Ressourcen, wie beispielsweise Schließfächer oder 3D-Drucker in einer Druckerfarm. Dabei kann entweder eine kurzfristige Identifikation abgegeben werden oder die Identifizierung dauerhaft gesetzt werden.

> **Hinweis:** Die Mermaid-Diagramme sind mit https://mermaid.live gerendert und als PNG exportiert und hier importiert worden, da BookStack keinen integrierten Mermaid Renderer besitzt.

URL und URN

Ressourcen in FabAccess werden durch einen Namen dargestellt. Um jedoch die Suche nach Ressourcen zu verbessern, können diese Referenzen als URN (Uniform Resource Name) oder URL (Uniform Resource Locator) geschrieben werden.

Die URN ist ein Identifikator ohne Bezug zum Space, während die URL auch den Space referenzieren kann, um einen Austausch von Ressourcen zu ermöglichen.

Genutzte Schemata in FabAccess

Folgende URN Schemata werden in FabAccess verwendet:

- von FabFire genutzte URNs:
  - Space Name: `urn:fabaccess:lab:{spacename}`
  - Space Info: `urn:fabaccess:lab:{spacename}\x00{instanceurl}`
    - dieser Wert wird aktuell nicht verwendet, muss jedoch ausgefüllt werden, damit die Konfiguration `bffh.dhall` valide ist!
- Ressourcen (Maschinen): `urn:fabaccess:resource:{machine id}` (ideal für QR Codes).

  > **Hinweis:** für das Scannen von NFC Tags kann stattdessen das folgende Schema verwendet werden (kompatibel sind z.B. Mifare NTags): `fabaccess://{spacename}/resource/{machine id}`. Siehe auch Ressourcen per NTag scannen