FabAccess

Apr 23, 2022

Contents:

FabAccess is still in a state of development and has no stable Release.

So there is no stability and backward compatibility between the versions. Feel free to help us testing our software stack but not expect a full functional software. We will inform you about breaking changes over our Zulip. Please subscript to it to stay up to date, because the server and client can get incompatible.

CHAPTER 1

Install FabAccess

FabAccess has an Server an Client structure.

So the Server and the Client must be installed seperatly.

1.1 Install Server

The Server can be installed over two recommended ways.

The easiest way is to use Docker with Docker-Compose.

To add youre own Plugins and Software you should use the Build way.

1.1.1 Build Server from Source

Install Dependencies

Ubuntu / Debian

1. `sudo apt update && sudo apt upgrade`
2. `sudo apt install curl && curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`
3. `sudo apt install libgsasl7-dev libssl-dev build-essential`
4. `sudo apt install git cmake clang capnproto`

Arch Linux

1. `sudo pacman -Syu`
2. `sudo pacman -S make cmake clang gsasl`

3. `sudo pacman -S git rust capnproto`

CentOS

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

FreeBSD

TODO

Build BFFH

Start new terminal - Rustup will not update path while install

1. `git clone https://gitlab.com/fabinfra/fabaccess/bffh.git --recursive`
2. `cd bffh`
3. (Optional) `git checkout development && git submodule update --remote` (Change `development` to wanted branch)
4. `cargo build --release`

1.1.2 Run Server with Docker

Docker Image can not run on armv6 (Raspberry Pi 1 or Raspberry Pi Zero)

1. Install Docker

   On Raspberry Pi: https://phoenixnap.com/kb/docker-on-raspberry-pi
2. Install Docker-Compose

   On Raspberry Pi: https://dev.to/elalemanyo/how-to-install-docker-and-docker-compose-on-raspberry-pi-1mo
3. Get Docker-Compose Files `git clone https://gitlab.com/fabinfra/fabaccess/dockercompose.git`

   The Dockerfile is in the root directory of the main repo docker-compose.yml is available in a seperate git repo
4. Edit config files in `config` folder to taste
5. Start Server with `docker-compose up -d`

To make it eaysier to apply youre changes in your config and keep the dockercompose uptodate, you should "fork" this respository.

Get Server Logs: `docker-compose logs`

1.2 Install Client

The Client communicates with the server over an API. So you can use any Client that support the current API Version of the Server.

We provide an Native Client for the following Platforms:

- Android
- iOS
- UWP

For MacOS and Linux(GTK) we will provide a Client later in our Development.

1.2.1 Get Client from Store

Hardware Requirements

Minimal Android Version: 5.0 (API Level 21)

Minimal iOS Version: 8.0

Store Links

- **Android**: https://play.google.com/store/apps/details?id=org.fab_infra.fabaccess
- **iOS**: https://apps.apple.com/us/app/fabaccess/id1531873374

1.2.2 Build Client from Source

On Windows

1. Install Visual Studio 2019 download Visual Studio
   - with Xamarin
   - with UWP
   - with .NET Desktop
2. Install GTKSharp for Windows download GTKSharp
3. Install capnproto

   3.1 If you have Chocolatey installed

   ```
   $ choco install capnproto
   ```

   3.2 else you can download it from here and add it to your PATH
4. Clone Borepin download Borepin

5. Load Borepin
6. Build Borepin

If Step 5. Build Borepin is failing because of GTKSharp, it could help to restart your PC.

Build GTK Project

1. Install mono download mono
2. Install mono, gtk-sharp, msbuild, nuget, capnproto 1.1 Debian based

```
$ apt install mono-complete, gtk-sharp2, libcanberra-gtk-module, nuget, capnproto,
↪ git
```

   1.2 ArchLinux based

```
$ pacman -S mono, mono-msbuild, gtk-sharp-2, nuget, capnproto
```

3. Update NuGet

```
$ nuget update -self
```

4. Clone Borepin

```
$ git clone https://gitlab.com/fabinfra/fabaccess/client.git --recurse-submodules
```

5. Build Borepin

```
$ cd client
$ nuget restore
$ msbuild -t:Borepin_GTK
```

6. Run Borepin

```
$ mono ./Borepin/Borepin.GTK/bin/Debug/Borepin.GTK.exe
```

You can also use Rider or monodevelop as an IDE for development on Borepin

macOS / iOS

1. Install Visual Studio for Mac
2. Install capnproto If you install capnp with Homebrew you may have to symlink the capnp binary into '/usr/local/bin', or bring it into your PATH another way.
3. Clone Borepin

```
$ git clone https://gitlab.com/fabinfra/fabaccess/client.git --recurse-submodules
```

4. Open in Visual Studio
5. Build

1.3 Installation Example for FabAccess Enviroment

1.3.1 Install on Ubuntu for "Dummies"

This description is how to compile and set up Diflouroborane on Ubuntu 20.04.3 LTS clean install. Other releases or distros might work as well. The process is quite lengthy but at the end you will have a FabAccess running to you needs. ... as I said: for complete dummies, if someone finds a better solution, please post suggestions on: https://fabaccess.zulipchat.com/#narrow/stream/255963-General/topic/Demo

1. Get your system up-to-date `sudo apt-get update && sudo apt-get upgrade`
2. Get rustup `sudo apt install curl curl --proto 'https' --tlsv1.2 -sSf https://sh.rustup.rs | sh` **restart the console**
3. Get capnproto, gsasl and git `sudo apt-get install gsasl sudo apt-get install capnproto` `sudo apt install git`
4. Create a target directory for BFFH there might be better places compared to where I created it, but it works... `mkdir BFFH` `cd BFFH`
5. Clone the Diflouroborane repository `git clone https://gitlab.com/fabinfra/fabaccess/bffh --recursive --branch main`
6. For compiling some dependencies were missing on Ubuntu `git submodule update --init sudo apt install libgsasl7-dev sudo apt install cmake sudo apt install libclang-dev` `sudo apt install libssl-dev`
7. Open the subdirectory and start compiling `cd bffh` `cargo build --release` **if the compiler prompts somthing like "error: linker 'cc' not found":** `sudo apt install build-essential cargo build --release`
8. Copy the configuration files (best done with the GUI filemanager of Ubuntu) copy files from "bffh/examples" paste them into "bffh/target/release/examples"
9. Install mosquitto MQTT broker Diflouroborane uses mosquitto MQTT broker to communicate with the Shellies. Starting from Ubuntu version 18.04, Mosquitto is already inside the official repositories. `sudo apt update -y && sudo apt install mosquitto mosquitto-clients -y`
10. Configuring mosquitto broker for some reason, starting with version 2.x mosquitto does not allow external communication via the broker per default. This needs to be changed via a config file:
11. Stop mosquitto `sudo service mosquitto stop`
12. Change into the "etc/mosquitto/" directory (lots of `cd ..` then `cd etc/mosquitto`)
13. Edit the configuration fil: `sudo nano mosquitto.conf` add: `listener 1883` `allow_anonymous true` Save (Strg-O) and close (Strg-X)
14. Enable mosquitto to start at each start of the system `sudo systemctl enable mosquitto`
15. Restart the system.

The BFFH-server can be found at the /target/release folder. Prior to starting the system you need to copy the files from `bffh/examples` to `bffh/target/release/examples` This ist best done by using the GUI filemanager.

To get at least minimum functionality the bffh.dhall should be modified. The lines:

```
-- , init_connections = [] : List { machine : Text, initiator : Text }
, init_connections = [{ machine = "Testmachine", initiator = "Initiator"
}]   , initiators = --{=}   { Initiator = { module = "Dummy", params = { uid =
"Testuser" } } }
```

should be modified to:

```
, init_connections = [] :  List { machine :  Text, initiator :  Text }      --,
init_connections = [{ machine = "Testmachine", initiator = "Initiator" }]
, initiators = {=}         --{ Initiator = { module = "Dummy", params = { uid =
"Testuser" } } }
```

To start the server change into the directory by using `cd target/release` and using:

`./diflouroborane -c examples/bffh.dhall --load examples` an then:

`./diflouroborane -c examples/bffh.dhall`

BUT prior to starting bffh, you should first configure the bffh.dhall file (see the "Use FabAcess" section).

CHAPTER 2

Configure FabAccess

To run FabAccess you have to configure the Server(BFFH). And add your Machines, Actors and Initiators.

In the current state of FabAccess we will no support database migration so all persistent data are in configuration files.

2.1 bffh.dhall

BFFH uses DHALL for Config-File structure BFFH uses RBAC for access control

BFFH Config is in `bffh.dhall` file.

2.1.1 General BFFH Config

`listens`

Contains the Addresses BFFH is listen for Connection for the API Default Port for BFFH is `59661`

Example:

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

`mqtt_url`

Contains the Address for the MQTT Server BFFH connects to **Example:**

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

`db_path`

Contains the Path for the internal Database BFFH uses. BFFH will create two files: `<db_path>` and `<db_path>-lock`. Make sure that BFFH has write access in the relevant directory **Example:**

```
db_path = "/tmp/bffh"
```

2.1.2 Permissions

BFFH uses a Path-style string as permission format, separated by ".". So for example `this.is.a.permission` consists of the parts `this`, `is`, `a` and `permission`. When requireing permissions, such as in machines you always need to give an exact permission, so for example `test.write`. When granting permissions, such as in roles you can either give an exact permission or you can use the two wildcards `*` and `+`. These wildcards behave similar to regex or bash wildcards:

- `*` grants all permissions in that subtree. So, `perms.read.*` will match for any of:
  - `perms.read`
  - `perms.read.machineA`
  - `perms.read.machineB`
  - `perms.read.machineC.manage`
- `+` grants all permissions below that one. So, `perms.read.+` will match for any of:
  - `perms.read.machineA`
  - `perms.read.machineB`
  - `perms.read.machineC.manage`
  - **but not** `perms.read`

Wildcards are probably most useful if you group you machines around them, e.g. your 3D-printers and your one bandsaw require:

1. Write permissions
   - `machines.printers.write.prusa.sl1`
   - `machines.printers.write.prusa.i3`
   - `machines.printers.write.anycubic`
   - `machines.bandsaws.write.bandsaw1`
2. Manage permissions
   - `machines.printers.manage.prusa.sl1`
   - `machines.printers.manage.prusa.i3`
   - `machines.printers.manage.anycubic`
   - `machines.bandsaws.manage.bandsaw1`
3. Admin permissions
   - `machines.printers`
     - For all printers

    - `machines.bandsaws`
        - For all bandsaws

And you then give roles permissions like so:

- Use any 3D printer:
    - `machines.printers.write.+`
- Only allow use of the "cheap" printers
    - `machines.printers.write.anycubic.*`
    - `machines.printers.write.prusa.i3`
- Allow managing of printers:
    - `machines.printers.+`
- Allow administrating printers:
    - `machines.printers.*`

This way if you buy a different anycubic and split the permissions to e.g.

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

It still works out.

2.1.3 Machine Config

`machines`

Contains list of machines

Machines have different perission levels to interact with:

- disclose: User can see the machine in machine list
- read: User can read information about the machine and there state
- write: User can use the machine
- manage: User can interact with the machine as Manager (Check, ForceFree, ForceTransfer)

Each machine must have an ID to reference the machine in other part of this config or over the API. And each machine must have a name.

Optional Information

To provide more information about the machine you can add it to the description or provid an external wiki link. Both attributes are only optional and do not need to be set.

Example:

```
machines =
{
    machine123 =
    {
        name = "Testmachine",
```

(continues on next page)

(continued from previous page)

```
        description = Some "A test machine",
        wiki = "https://someurl"

        disclose = "lab.test.read",
        read = "lab.test.read",
        write = "lab.test.write",
        manage = "lab.test.admin"
    }
}
```

“machine123” is in this case the “Machine-ID”

2.1.4 Roles Config

The roles are configured in the bffh.dhall. If the file “roles.toml” is existing in the directory, it can be deleted and can’t be used to manage roles.

roles

Contains list of roles

Roles have a list of permission and can be inherited. Permission can be wildcard in permission list.

Example:

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

2.1.5 Actors Config

actors

Contains list of actors Actors are defined by a module and one or more paramters

Currenty supported actors:

Shelly Actor

This actor connects BFFH over an MQTT-Server to an shelly device.

You need to set the `topic` parameter of the Shelly to the Shelly specific MQTT-Topic.

Find shelly topic here

Example:

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

"Shelly_123" is in this case the "Actor-ID".

Process Actor

This actor makes it possible for you to connect your own Devices to BFFH.

`cmd` = Path of executable

`args` = Arguments for executable

Example:

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

actor_connections

Connects the actor with a machine A machine can have multiple actors

Use the "Machine-ID" and "Actor-ID". **Example:**

```
actor_connections =
[
    { machine = "Testmachine", actor = "Shelly_1234" },
    { machine = "Another", actor = "Bash" },
```

(continues on next page)

(continued from previous page)

```
    { machine = "Yetmore", actor = "Bash2" }
]
```

2.2 user.toml

CHAPTER 3

Use FabAccess

FabAccess is highly customisable so you can use FabAccess the way you like to.

But to explain our Features we will documentated some best Practices.

3.1 QR-Codes on Machines

To imporve the selection of the right machine for youre users you can apply a QR-Code label on the machine.

The QR-Code must have the following content:

`urn:fabaccess:resource:{machine id}`

3.2 FabAccess Setup - Step By Step

This document provides a step by step Instruction on how to get FabAcess running. At the end of this description you will have:

- 1 or more Shellies registered to you system
- 1 or more users registered to your system
- QR-Codes generated to acess a machine
- 1 Shelly configured as a door-opener
- 1 Shelly configured to identify if a machine is just switched on or realy running (TO-DO)

Step 1 Installing the BFFH-Server

there are multiple ways to install the BFFH server. This can bei either done via

- docker - see docker installation document
- installing from source - see installing from source documentation

- installing on Ubuntu for dummies

Step 2 Installing the FabAccess App

get the App via Apple Store or Google Apps.

Step 3 Connect the App and the Server

First you need to find the IP of the server. This can be done by typing `ip a` on the console of the system where the BFFH-Server is running. Use the adress listed under BROADCAST.

Start the server. If you are using the docker, this is done by using `docker-compose up -d`. If you compiled the server on your system this is done by entering `./diflouroborane -c examples/bffh.dhall --load examples` and then `./diflouroborane -c examples/bffh.dhall`. You will see some debug information, with probably some warnings.

Open the App. You will be asked to connect to a Host. Tap "DEMO HOST ADRESS" and change the IP to the IP of your Server, do not change the port number (everything after the IP. This should look like `192.168.1.15:59661`). Tap "SELECT HOST".

You will be asked to sign in. For Version 0.2 only the Option "LOGIN WITH PASSWORD" ist available. Use `Testuser` and the passwort `secret` to log in.

You will find an overview of the installed machines including the option "SCAN QR-CODE". Next step is setting up you machines so they can be switched on an off.

Step 4 Prepare your Shellies

as long as your Shelly has not been given the credentials for a WLAN, it will create an access point (AP) for configuration when connected to the supply voltage. This AP will appear in your list of WLAN. Connect to this Shelly-AP and connect to `192.168.33.1` in your browser. A configuration page should appear. If your Shelly is already connected to your WLAN, you must find the assigned IP-Adress (e.g. by looking into your router). Enter this IP Adress in your browser and you will get the configuration page.

Shelly MQTT Client setup

goto "Internet & Security" -> "Advanced - Developer Settings" enable "MQTT" enter the IP-Adress from your Server in the field "IP-Adress" As we did not define MQTT credentials in mosquitto yet, no creadentials need to be filled in. To find the "ID" of your Shelly activate "Use custom MQTT prefix" (but do not change it!). This should be somthing like: `shelly1-123456789ABC` for a Shelly 1 `shelly1pm-123456` for a Shelly 1PM note this ID for later **- save - re-check the settings!**

Shelly WLAN Client setup

goto "Internet & Security" -> "WIFI MODE - CLIENT" Set WLAN Credentials

Adding a Shelly to your server To understand the underlaying concept of actors and machines, please see the "configuration part" of the documentation. Four our example we will assume we have one actor (shelly) per machine.

Tip Prior to modifying the configuration files the proper working of the MQTT broker should be tested. To test the broker it is the best to use a second (linux) computer with a different IP adress. To test if the broker allows access from an external IP address open a MQTT subscriber on the second computer by typing `mosquitto_sub -h 192.168.1.15 -t /test/topic` (change the IP adress to the adress of your server). Use `mosquitto_pub -h localhost -t /test/topic -m "Hallo from BFFH-Server!"` to send a message to the other computer. If the message appears, everything is ok. When not, this should be first solved, as a connection to the shellies will not be possible this way. If you are interested in communication between the shellies and the BFFH-Server you can use `mosquitto_sub -h 192.168.1.15 -t shellies/#` (change the IP adress to your needs). You will see some values popping op from time to time.

Configure Diflouroborane Open the file "bffh.dhall" in the GUI Editor (just by double-clicking it) or use `nano bffh.dhall` in your console.

Link the server to the MQTT-broker find the line which starts with `, listens`. You will find three lines stating addresses. The third address needs to be changed to the adress of your MQTT broker (most likely the IP adress of your BFFH server)

First you have to make your "actors" (in our case the Shellies) know to the system. Go to the line where it starts with `, actors =` and after the `{` you can enter your Shelly with `shelly1-123456789ABC = { module = "Shelly", params = {=}}` The ID of the Shelly should match the ID of your Shelly. Here you can enter as many actors as you want, each separated by a `,`.

Now you have to link a "machine" to an "actor". Go to the line starting with `{actors_connections =` and after the following `[` you add `{ machine = "Identifier-of-your-Machine", actor = "shelly1-E8DB84A1CFF4" }` using your own Name-of-your-Machine and the Shelly-ID of the related actor.

Now you have to set the "access-permissions" to your "machine". Go to the line starting with `, machines =`. and after the `{` you can add a machine: `Identifier-of-your-Machine = { description = Some "I am your first Testmachine" , disclose = "lab.test.read" , manage = "lab.test.admin" , name = "Name of the Machine" , read = "lab.test.read" , write = "lab.test.write" },`

Please be aware that "Identifier-of-your-Machine" is the internal ID for BFFH. The name of the machine shown in the App will be "Name of the Machine". The given permissions are ok to start with (if you did not change the roles of the Testuser). To find out more about the permission concept see the "configuration" part of the documentation.

- save (if you are using nano, this will be Ctrl-O)

-restart the BFFH-server Important every time you change the bffh.dhal you need to reload the settings (otherwise the App will not connect to the server on restart): `./diflouroborane -c examples/bffh.dhall --load examples` and restart start Diflouroborane: `./diflouroborane -c examples/bffh.dhall`

Open the App, an you should see the newly created machine in the list. By tapping "USE" you will activate the machine (Shelly will click, the MQTT-listener should promp an "on"), by tapping "GIVEBACK" you will deactivat the machine.

Creating a QR-Code for your machine A QR code allows users to directly enter the UI of the machine, where the machine can be used or given back. The QR code should contain the following content: `urn:fabaccess:resource:{MachineID}` e.g. `urn:fabaccess:resource:Identifyer-of-your-Machine`

QR-Codes can be generated on various pages in the internet (e.g. https://www.qrcode-generator.de), the "Type" of the QR code should be "Text". The generated code can be directly scanned by the FabAccess App in the machine overview.

Adding a user Adding a user to the system consists of two steps

- creating the user
- provide permissions

Users are defined in the file users.toml. To add a user simply add `[Name-of-the-User] roles = ["Name-of-a-role/internal", "Name-of-another-role/internal"]  priority = 0 passwd = "the-chosen-password" noot = "whatever-this-means"` Adding users or changing existing users does NOT require to restart the system (tested?)

The permissions of the user are defined by the linked roles. The roles are defined in the file bffh.dhall. Open the file bffh.dhall an find the line starting with `, roles =` The concept of the role management is described in the "configuration" part of the documentation. To keep it simple we create a role called "ChainsawUser" `ChainsawUser = { permissions = [ "lab.machines.chainsaw.write"` - allows the user to use the machine `, "lab.machines.chainsaw.read"`- allows the user to read see the status of the machine `, "lab.machines.chainsaw.disclose"` - allows the user to see the machine in the machine overview `]`

If a user assinged to this role uses the chainsaw, no other user is able to use it until this user gives the chainsaw back. To unlock the machine from the user, admin permissions are needed. So there could be an admin role like `ChainsawAdmin ={ parents = ["ChainsawUser"]` - inherits all the permissions of the ChainsawUser `,` `permissions = ["lab.machines.chainsaw.admin"]` - addinional admin permissions `}`

The `machine` should be defined as: `Identifier-of-your-Chainsaw = { description = Some "Beware of Freddy!" , disclose = "lab.machine.chainsaw.disclose" , manage = "lab.machine.chainsaw.admin" , name = "Chainsaw" , read = "lab.machine.chainsaw.read" , write = "lab.machine.chainsaw.write" },`

If a user is asigned to “ChainsawUser/internal” he/she will be able to see and used the chainsaw in FabAccess.

Using a Shelly as a door opener (electronic wise) In version 0.2 a door opener functionality is not implemented. The specific behaviour of a door opener is, to activate a door openeing relais only for a few seconds. This behaviour is not yet implemented in FabAccess, but there is decent way to implement it by other means. The simple Shellies (1, 1pm, 2.5) have an internal timer “AUTO-OFF” which can be set. To use this timer you have to access the settings of the Shelly via a browser on your computer. To do so, you have to know the IP adress your Shelly is assinged to. This can normally found out in the router of your Wifi. By entering this IP adress in your browser you will access the main menu of your Shelly.

Go to “Timer” and set the “AUTO-OFF” to e.g. 3 seconds. Define a machine called “door” in the bffh.dhall

- define the actor: `shelly1-123456789ABC = { module = "Shelly", params = {=}}`
- define the machine: `{ machine = "door", actor = "shelly1-123456789ABC" }`
- set permissions for the machine: `door = { description = Some "close it firmly" , disclose = "lab.door.disclose" , manage = "lab.door.admin" , name = "Door to the Lab" , read = "lab.door.read" , write = "lab.door.write" },`
- create a role having ALL permissions to the door `DoorUser = { permissions = [ "lab.door.write"` - allows the user to use the door `,` `"lab.door.read"`- allows the user to read see the status of the door `,` `"lab.door.disclose"` - allows the user to see the machine in the machine overview `,` `"lab.door.admin" ]`
- assign the role DoorUser/internal to all users

It is imporatant all users have admin aka manage permissions, as the request to open the door by a user, thet the door “in Use” by this user. The door can only be re-activated when the previous user “un-uses” the door or if an othe user can “force free” the door prior to using the door hin/herself. **Note** in this special case, where all users will need admin capabilities the role could also contain only the permission `lab.door.use` and all permissions (disclos, manage, read, write) assigned to the machine would simply match `lab.door.use` (e.g. disclose = “lab.door.use”‘).

**Identify if a machine is just switched on or realy running (TO-DO)