

# 03.12.2019 // Roseguarden

Repositories (v2): <https://gitlab.com/roseguarden>

Demo: <https://roseguarden.fabba.space/dashboard>

## Vorbetrachtungen

- Ziel ist ein möglichst einfaches/übersichtliches System zu erstellen
- Möglichst wenig externe Infrastrukturabhängigkeiten bzw. Voraussetzung
  - keine Broker
  - keine statisch vorausgesetzten Datenbanken
  - keine als notwendig vorausgesetzte / aufbauende Softwareinstallationen  
=> diese machen das System unnötig kompliziert für Einsteiger (Abschreckung)
- Möglichst gute Anpassbarkeit und Erweiterbarkeit (Opt in Gedanke)
- Möglichst viele Schnittstellen als Module (Optionen schaffen)
- Bibliothekenabhängigkeiten sind ok (alles was integriert)
- Möglichst einfache Integration von externen Zugriff ermöglichen (Schnittstelle nach außen ist genauso wichtig wie gute Schnittstellen intern)
- Standards vor Eigenentwicklung
  - möglichst wenig selber entwickeln (zu wenig Manpower)
  - möglichst beliebte Bibliotheken, Sprachen, und Werkzeuge verwenden (sollte maintained sein/bleiben)
  - Komplexität sollte gering gehalten werden
- Möglichst einfache Bedienung für nicht Programmierer ist sehr wichtig (gutes Frontend)
- Einfache Installation und Wartung für Nutzer/Admins ist ebenfalls sehr wichtig
- Möglichst günstig und weit verbreitet Hardware einsetzen
  - kein Raspberry Pi (Erfahrung aus v1) da Probleme mit Einrichten / Automatisierung / kaputte Dateisysteme / Hardwareanbindung / Hoher Stromverbrauch / Kosten (durch Zusatzkomponenten, evt. Rasp Zero W ?!)
  - Raspberry Pi eher gern als Server
  - Mikrocontroller (ESP8266, ESP32, Arduino) zu bevorzugen
- Gut aufbereitete Doku (Doku included)

## Projektinterne Werkzeuge

Im Folgenden werden die aktuell verwendeten digitalen Werkzeuge aufgeführt sowie eine kurze Begründung für deren Entscheidung.

# Python

## Für die **Backend-Entwicklung**

- Python als weit verbreitete Sprache
- Popularität siehe Bild



(Quelle: <https://entwickler.de/online/development/top-10-programmiersprachen-redmonk-tiobe-pypl-579886370.html>, gern mal zum direkt anschauen, die Zahlen bzw. Abstände der Sprachen sind sehr interessant, hier sind die 3 wichtigsten Rankings (Methoden) aufgelistet)

- Viele Leute hatten bereits Kontakt (persönliche Erfahrung)
- allgemein wird eine flache Lernkurve wahrgenommen (persönliche Erfahrung)
- einfache Installation mit anaconda / pip
- auf allen OS Verfügbar (Linux, Windows, Mac)
- riesiges Portfolio an Bibliotheken (siehe Ziel: möglichst wenig selber programmieren)
  - requests - HTTPS Anbindung an REST-APIs und Co
  - python-ldap - LDAP Anbindung
  - fintech - Anbindung an SEPA, DATEV,
  - python-openhab - Anbindung an OpenHab
  - Odoo Client Library - Anbindung an Odoo
  - Hardwareansteuerung - RFID-Reader, Relays, etc
  - Flask - Website-Framework
- Andere Programmiersprachen in Python-App sehr gut integrierbar
  - Vorteil von Python das Kombination mit anderen Sprachen sehr einfach möglich ist (sozusagen seamless), wenn gewünscht
  - z.B. Plugins anbieten die in anderen Sprachen geschrieben sind (kein Hauptziel, vielleicht später da auf ersten Blick kompliziert)
  - siehe <https://wiki.python.org/moin/IntegratingPythonWithOtherLanguages>
  - Integration von Java-Code über Japp
  - Integration von C/C++ Code über Cython / Ctypes
  - Integration von C#-Code über IronPython
  - Integration von Rust-Code über PyO3
- Python lässt das dynamische laden von Modulen (Z.b. als Plugins sehr einfach umsetzen), ohne overhead (einfach Quellcode-Dateien in Ordner kopieren)

## **Nebenbemerkung Javascript für die Frontend-Entwicklung**

- Für separate Frontendentwicklung gibt es neben plain HTML/CSS (meist generierte Seiten) faktisch keine nennenswerten Vertreter neben JavaScript/TrueScript (WebAssembly wird meist nur als Zusatz verwendet und daher hier erstmal außen vor) >> JavaScript
- Typescript umstieg natürlich möglich/offen (Typensicherheit etc.)

## Flask

Python-Framework für die **Backend-Entwicklung**

- Siehe: [blog.miguelgrinberg.com](http://blog.miguelgrinberg.com)
- Ziel: Vereinfachung der Entwicklung, bestehende Bibliotheken nutzen
- Anwendung wird übersichtlicher und einfacher zu verstehen
- Einfacher/Schlanker als Alternative Django und dabei ähnliche Verbreitung
- Viele Erweiterungen:
  - Migration von MySQL, MariaDB, PostgreSQL, MSSql oder SQLite je nach Wunsch
  - Flask-SQLAlchemy - Datenbankintegration
  - Flask-JWT - Sicherheitstoken
  - Flask-Mail - Integration von eMail-Versand
  - Flask-Upload - Datei-Upload handling
  - Flask-Rabbitmq - Integration von RabbitMQ
  - Flask-MQTT - Integration von MQTT-Nachrichten
  - Flask-ZMQ - Integration von ZeroMQ Sockets

## Vue

Für die **Frontend-Entwicklung**

- <https://vuejs.org>
- JavaScript-Bibliothek zur Datenverwaltung (DataBinding) und als Hilfestellung, de facto-Standard ein Framework zu nutzen
- Die Standard-Player mit großer Community sind Angular (Google), React (Facebook) und Vue (Indie)
- Gute Zusammenfassung siehe: <https://dzone.com/articles/react-vs-angular-vs-vuejs-a-complete-comparison-gu>
- Kleines Projekt mit kleinem Team <= "Vue is ideal for a small team and a small project. If your app seems to be large and has significant future expansion plan, pick React or Angular."

## Vuetify

## Für die **Frontend-Entwicklung**

- Ist eine Komponenten-Bibliothek auf Basis von Material Design für Vue
- Enthält GUI-Elemente: Formulare, Kalender, Tabellen ...
- Internationalisierung included
- Anpassung von Farbe Style sehr einfach
- Sehr gute Doku mit vielen Beispielen
- möglichst wenig selbst entwickeln
- Siehe Beispiel für [Kalender Komponente](#)
- Beschleunigt Entwicklung
- Mobile und Browser-optimiert
- ggf. keine separate Handy-App mehr notwendig (wenn ja, sehr einfach per Electron)

## HTTP/S

Als Schnittstelle nach außen

- Möglichst nur **eine** Schnittstelle Maintainen für möglichst geringen Aufwand
- Für Frontend (Website + Apps) ist HTTP/S die meist verbreitetste Technologie (je nach Frontend-Entwicklung zwingend notwendig, alternativ Websockets nutzen und direkt abkapseln, Entwicklungsaufwand?)
- Keine zusätzliche Installation von Brokern notwendig (wie MQTT/RabbitMQ, da möglichst geringe Infrastrukturabhängigkeit / Komplexität / Wartung gewünscht)
- Auch in Mikrocontrollern einfach umsetzbar (ESP8266, ESP32, Arduino)
  - ZMQ erfüllt dieses Kriterium nicht
  - MQTT erfüllt diese je nach Hardware
  - alternativ selbst entwickeln?
  - kennt jemand Bibliotheken für Arduino und ESP?
- Problem der NAT (Netzwerkübertragung in eigener Infrastruktur) bzw. Firewalls (HTTP generell sehr einfach durchzukriegen), bidirektionale Schnittstellen schwerer, IP-finden zulassen, Sicherheitsrelevanz?
- Fast alle Web-Dienste bieten eine HTTPS-Schnittstelle (REST) an, dadurch sehr einfache Anbindung externer Tools (ohne strenge Kopplung)
  - Nextcloud
  - Calendar
  - Chats wie RocketChat
  - Trello/Openproject
  - OpenHab/HomeAssistant
  - Odoo
- Durch Schaffung einer eigenen HTTPS Schnittstelle (z.B. REST) kann das System auch nach außen/extern ankoppelbar/integrierbar gemacht werden (Andere

Dienste, Skripte)

- Verschlüsselt / Sicherheit hoch
- Verbreitung sehr hoch
- Einfach zu verwenden
- Es können natürlich auch weitere Protokolle angebunden werden, wie bspw.:
  - MQTT
  - Websockets
  - ZMQ
  - RabbitMQ
  - meist als Flask-Extension verfügbar (siehe Punt Flask (Backend-Entwicklung))
- Frage ist eher, welche ist der Erstwunsch bzw. welche als Hauptschnittstelle

## Eigenentwicklung von Hardware

- Eigenentwicklung von Hardware für Endkunden ist in Deutschland teuer und bürokratisch
  - **CE Zertifizierung**
    - relativ einfach
    - geringe Kosten
    - Verantwortlichkeiten/Rechtliches teilweise unklar
  - **Sicherheitsbestimmung 230V**
    - mittel kompliziert
    - geringe Kosten
    - Verantwortlichkeiten/Rechtliches unklar
  - **VerpackungsG**
    - einfach
    - mittlere Kosten
    - Verantwortlichkeiten/Rechtliches klar
  - **ElektroSchrottVerordnung ElektroG**
    - kompliziert
    - hohe Kosten, einmalig + jährlich
    - Containerlotterie
    - Rechtliches unklar
  - **ROHS Konfirmität**
    - einfach
    - geringe Kosten
    - Rechtliches relativ klar
- Komponenten lieber als Set zusammenstellen
- Hilfe zur Selbsthilfe anbieten (Leiterplatten, Bausätze?!)

- Allgemein liegt die Hardwareverantwortung eher in den jeweiligen Gruppen was auch meist so gewünscht ist
- Alternative B2B-Verkauf (keine Consumern, nur z.B. Vereine) da es dann einfacher ist. Es ist trotzdem noch aufwändig.

# Systemdesign

## Koordinator

- Zentraler Server mit Anbindung an verteilte Systeme
  - **Vairante A: Lokal** (z.B. Raspberry Pi Server)
    - Lokal im Space und kann ggf. nur dort erreicht werden
    - Ansprache über IP statt Domain möglich
    - Selbstgeneriertes Zertifikat für HTTPS notwendig
    - Zugriff benötigt DynDNS
  - **Varinate B: Gemieteter/eigener Server**
    - Domain angebunden
    - letsencrypt Zertifikate
    - kann von überall erreicht werden
    - Serversicherheit und Datenhoheit je nach Gruppe anders priorisiert (persönliche Wahrnehmung)
- Einsicht des Kontos über Website (Frontend) ggf. App
- Integration externer Hardware über Backend möglich
- Administration der User Nodes zentral über Website (Frontend)
- Permissions als Gruppen verwaltet durch Server
- Gruppenverwaltung (Zugriffe, Zugänge) angedacht, verwaltet durch Server

## Nodes

- Verteilte Eingabe/Steuer-Geräte (Knoten/Nodes)
- Beispiele:
  - RFID-Reader fürs Büro zum Verwalten von Karten
  - Terminal zum Aufladen der Karte mit Guthaben
  - Terminal zum Auslesen von Guthaben nach Eingabe einer Pin
  - Türschlossansteuerung ggf. mit Öffnungsdetection
  - Relais zum An/Freischalten von Geräten
- Nodes verbunden per Lan oder Wifi an Server (lokal/extern)
- Knoten laufen autonom und sind möglichst dumm (werden vom Server gesteuert)
- Knoten laufen mit möglichst geringer Hardwareanforderung (ESP, kein Linux, 4MB Ram, Arduino je nach Typ)
- Knotenkosten gering:

- Relay und RFID-Knoten < 20€ (mit display < 50€)
- Türschlossknoten > 50€
- Knotenhardware ggf. selbst entwickeln (siehe Hardware bzw. Bedenken oben)
- Knoten ggf. mit Wifi-Mesh-Funktion (siehe ESP32-Mesh :

<https://docs.espressif.com/projects/esp-idf/en/latest/api-guides/mesh.html>)

## Authentifikation

- Authentifikation über RFID-Karten + optionaler Pin-Auth (ggf. über Smartphone mit Pin-Auth)
  - 13,56 MHz Mifare favorisiert
    - Desfire sollten auch gehen
  - Gewünscht nur UID auswerten, da andere Karten dann wiederverwendet werden können wie Studentenausweis
    - dann aber 2ter Faktor notwendig!
  - Smartphone NFS evtl. später (haben hier keine Erfahrung damit, nicht niederschwellig da vom Smartphone abhängig > Parallel anzubieten)
  - 125kHz evtl. später integrieren

## Beispiele des Datenflusses

### Freischalten neuer Karten im Büro

- Heartbeat überprüft alle 10 Minuten ob noch eine Verbindung zum Server besteht (am besten mit Alarm bei Ausfall)
- Wenn Karte erkannt wurde, Senden der Kartennummer etc. an Server (sofort)
- Zuordnung im Frontend (Zuweisung nach Gruppen)

### Gerät freigeben

- Heartbeat überprüft alle 10 Minuten ob noch eine Verbindung zum Server besteht (am besten mit Alarm bei Ausfall)
- Wenn Karte erkannt wurde folgt sofortiges Senden der Authentifizierungsanfrage an Server (ggf. erweitert um Pinabfrage)
- Server prüft und gibt direkt Rückmeldung (bspw. über ein LCD Display und/oder ein Audiosignal)
- Knoten schaltet Gerät frei (Relais)

### Öffnen einer Tür

(Reader und Türsteuerung an verschiedenen Orten / Knoten)

- Heartbeat überprüft alle 10 Minuten ob noch eine Verbindung zum Server besteht (am besten mit Alarm bei Ausfall)
- Wenn Karte erkannt wurde folgt sofortiges Senden der Authentifizierungsanfrage an Server (ggf. erweitert um Pinabfrage)
- **Variante A:** Server sendet direkt an Steckdose (zWave etc.) den Freischaltbefehl
  - hat ggf. das Problem der Erreichbarkeit und Firewalls auf beiden seiten
- **Variante B:** Server speichert anfrage und wartet auf Pollen der eigenen Hardware/Knoten => Geräteüberwachungs-Node (pollt z.B. alle 0.5s)
  - Ggf. langsamer, dafür nur ein zentraler "Master"

## Getränke kaufen

- Terminal mit RFID Lesegerät neben der Getränkequelle
- Wenn Karte erkannt und die PIN korrekt ist (bei 2FA) wird Guthaben vom Konto auf der Serverseite abgebucht

## roseguarden Backend

- Backend in Python
- Flask als Framework
- Entwicklungsumgebung VS Code (mit Anaconda), einfache Einrichtung (andere mögliche, sublime, vim etc)
- Modulares System (Pluginsystem)
  - Problemstellungen abstrahiert und modularisiert nach "Workspaces"
  - Beispiele für Workspaces: UserAdministration, Spaces, Permissions etc.
  - Workspaces (Plugins) werden als einfacher Ordner mit Python-Files (nach einer Vorlage) einfach in das Projekt kopiert und werden dynamisch beim start geladen
  - Umsetzung als Klassen die geladen werden
  - Jeder Workspace (Plugin) kann Seiten (Pages), Daten (DataViews) und Aktionen (Actions) hinzufügen
- Supervisor zum automatischen Update, Migration der Datenbank, Backup
- Backend generiert keine Seiten sondern stellt Daten/Logik bereit (Frontend zur Darstellung unabhängig)

## roseguarden Frontend

- Frontend mit Vue und Vuetify (Material GUI Komponenten)
- Separat von Backend (kann separat Entwickelt, getestet werden)
- Entwicklungsumgebung VS Code (mit Anaconda), das gleiche wie Backend
- einfache Installation dank npm
- Einfache Struktur dank nuxt (einfach dateien kopieren)



- Styling (Farben, Icons, etc.), sehr einfach möglich
- Frontend beinhaltet keine Logik sondern bekommt Befehle und Daten durch Backend
- Siehe Demo + Repo

## roseguarden Hardware

- Möglichst günstig und weit verbreitet Hardware einsetzen
- Wir nutzen [Olimex-Module](#) (Bulgarien)

- [Gateway mit Relay](#) 26€



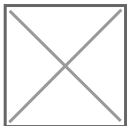
- [RFID ID Reader](#) 13€



- [Power over Ethernet Knoten](#) 25€



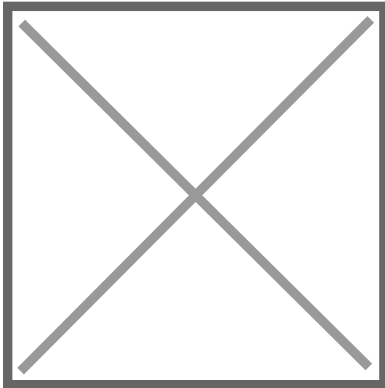
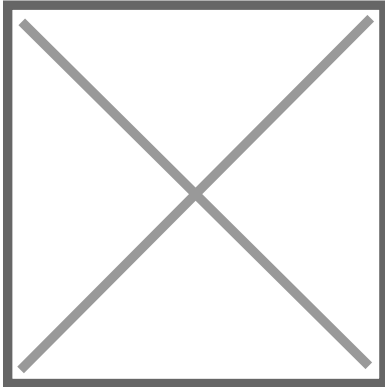
- [Display mit Touch](#) 15€



- jeweils auch mit externer Wifi-Antenne möglich
- ESP32 immer auch mit Akku möglich
- Andere Module Anschließbar
- Basis ist die ESP32 Mikrocontroller-Familie
- Eigene Hardware (Leiterplatte entwickelt, für Eigenbedarf)
- Integration anderer Hardware möglich

## Weitere Hardware

- [Bildschirm mit Touch](#) (Nextion) ab 20€
  - Aktuell wird dieser Touchscreen von uns verwendet
  - Grund dafür ist, dass die Entwicklung eines Frontends dank der mitgelieferten Software sehr einfach mit einer GUI erfolgen kann --> kürzere Entwicklungszeit



- [ABUS Funk-Türschlossantrieb HomeTec Pro CFA3000](#) @100€
  - Da es in unseren Räumen schon mehrfach zu Einbrüchen kam, ist der Wunsch nach abschlossenen Türen aufgekommen
  - Das Makerspace aus Oldenburg hat mir (Kevin) den Tipp gegeben dieses Gerät zu verwenden.
  - Der Schlossantrieb hat seitlich zwei elektrische Kontakte mit denen man den Antrieb steuern kann
  - Weiterer Vorteil ist hier auch, dass es im Notfall immer auch manuell betrieben werden kann. Selbst wenn der Strom ausfällt



- Domra Türöffner (Summer) ca. 20€



## Interessante Hardware für später

- zWave Hardware
- HomeKit Hardware
- ...

# roseguarden Firmware für die eigenentwickelte Hardware

- C++ für ESP32
- Entwicklungsumgebung Eclipse mit Espressif-IDF (VS Code denkbar, bisher keine Arduino IDE)
- eigene VM vorhanden (kann gerne geteilt werden)
- Bibliotheken von Espressif (JSON-Parser, Wifi, Bluetooth, etc.)
- FreeRTOS als Mikrobetriebssystem
- Verwendung bestehender Bibliotheken
- Modularer Ansatz (Kapselung in Klassen)
- ...

## Berücksichtigte/Angedachte UseCases (Features)

- Zugriff und Rechte als Gruppen
- Benutzerverwaltung ggf. Vereinsmitglieder
- Zugangskontrolle Türöffner
- Gerätefreischaltung angedacht
- Nachrichtensystem (inklusive Alerts für Admins)
- Buchungssystem?
- Buchhaltung (DATEV)?
- Geld aufladen?
- Mitgliederbeiträge verwalten (Fintech, SEPA)
- ...
- tbd

## Offene Punkte

- Lizenz Software (Wunsch GPLv3)
- Lizenz Hardware (Open Hardware Lizenz)
- Datenschutz / DSGVO?
- Rechtlich saubere Struktur notwendig? Verein? etc.
- Bündeln der Kapas, nur wie?
- Wie werden Entscheidungen getroffen
  - Gutmütiger Diktator/ Managergremium
  - Basisdemokratische Abstimmungen vs Abstimmung in Gremien
  - Wie alle auf einen Wissensstand bringen
  - Systemisches Konsensieren (<http://www.sk-prinzip.eu/das-sk-prinzip/>)

# Historie von roseguarden

- Roseguarden entwickelt als Version 1 (<https://github.com/mdrobisch/roseguarden>)
  - Backend: Python + Flask
  - Frontend: AngularJS + Bootstrap
  - Hardware: Raspberry Pi + Relays + RFID-Reader
- Produktivsystem seit 2014 in Dresden
- Mit verschiedensten Anwendern in Kontakt (Spaces, Fablabs, Vereine, Firmen z.B. Lager, Event-Veranstalter)
- Vor 4 Jahren 1. Versuch Systeme zu vereinheitlichen und Leute zu finden, die mitentwickeln (Vernetzungstreffen des Verbund Offener Werkstätten e.V. 2015 in Dresden)
- Einreichung im Prototypefund (als Gruppe 3 Personen, und einzeln)
  - Feedback/Fazit: Kombi aus Hardware und Software zu kompliziert und nicht gut vereinheitlichbar,
  - woanders Förderbar: Anstiftung, BMBF direkt?!
- Seit Jan. 2019 Planung um Umsetzung einer Version 2 in einer Gruppe von 3 Leuten

---

Version #2

Erstellt: 13 Oktober 2024 01:11:02 von Mario Voigt (Stadtfabrikanten e.V.)

Zuletzt aktualisiert: 25 Februar 2025 21:22:13 von Mario Voigt (Stadtfabrikanten e.V.)