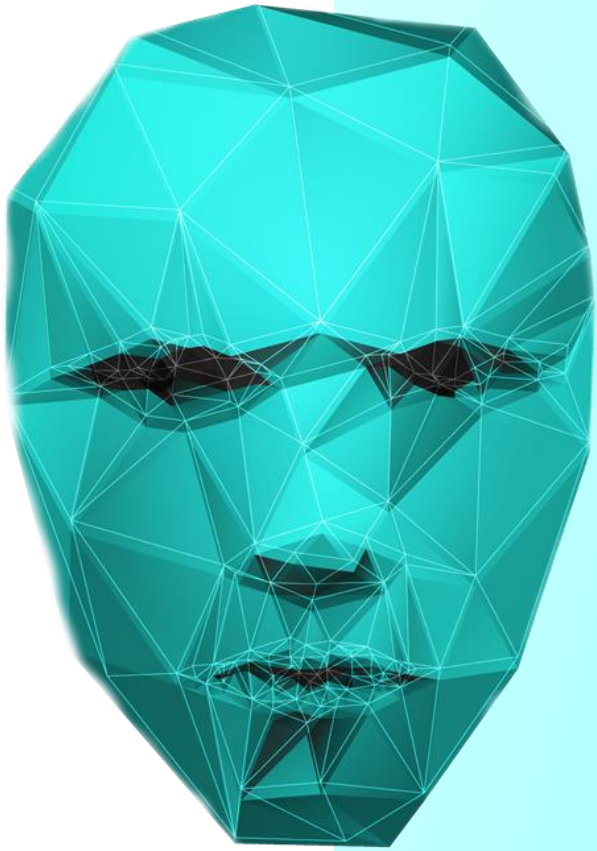
Gaia Smarthome



UBQ Projekt SS2021

Leia Lederer (358096)

Marcel Henning (379495)

Marvin Henning (379508)

Table of Contents

Table of Contents 1
Einleitung 3
Aufgabenstellung 3
Entwicklungsumgebung 3
- VSCode 3
- IntelliJ IDEA 3

Projektplanung 3
- Projektidee 3
- Ziel des Projekts 4
- Grober Projektentwurf 4
- Umsetzungsübersicht 4
- Zeitplan 4

Design 5
- Designfindung 5
- Elemente 6
  - Obere Leiste 6
  - Gruppen-Menü 6
  - Content 6
    - Widgets: 6
  - Untere Navigationsleiste 6
- Settingspage 7
- Umsetzung 7
  - Angular Material 7
  - Flex-Layout 8

Raspberry Pi 8
Shelly 1 8
Temperatursenor ESP8266 mit DS18b20 9
- Motivation 9
- Verkabelung 9
  - Sketch für den ESP8266 10

Angular 12
Docker 12
MQTT 13
MongoDB 13
Tasmota 14
Client 14
Server 15
- UML 16
- Embedded MQTT Broker 16
- Automatisiertes Anlegen von Geräten 16
  - Tasmota 16
  - Generalisierung 17

Datenmodell 17
Deployment 18
- Wie wird aus der Anwendung ein Image? 18
- GitHub Actions 19
  - Workflow 19
  - Dockerhub 20

Installation 20
- Vorraussetzungen 20
- Ablauf 20

Erreichtes Ziel 23
Unsere Funktionalitäten 23
- Basis 23
  - Server 23
  - Webapplikation + App 23
  - Geräte hinzufügen 23
  - Geräte ansteuern 24
  - Datenbank 24
- Dashboard 24
  - Individuell anpassbares Dashboard 24
  - Widgets 24
    - Temperatur 24
    - Wetter 24
    - Licht 25

Kategorien 25
Gruppen 25
Einstellungen 25
Serveradresse 26
Themes 26
Sprache 26
Was wir uns noch gewünscht hätten 26
Alexa Integration 26
Benutzerübersicht 26
Webservice 26
Zeitjournal 27
Zeitjournal der einzelnen Teammitglieder 27

Einleitung

Aufgabenstellung

Im Rahmen des Moduls Ubiquitous/Pervasive Computing welches von Professor Knauth im WS20/21 gehalten wurde, sollten die Studierenden die Vorlesungsinhalte anhand eines persönlichen praktischen Projekts vertiefen und anwenden.

Für die Durchführung des Projekts fanden wir uns in Kleingruppen zusammen. Unsere Gruppe setzte sich zusammen aus Marvin Klein (Informatik), Marcel Henning (Informatik) und Leia Lederer (Wirtschaftsinformatik). Jede Gruppe konnte daraufhin ein für sie passendes und interessantes Thema aussuchen.

Das von uns gewählte Thema konzentrierte sich darauf, dass wir eine Smarthome-Plattform schaffen wollten, die selbst unerfahrenen Usern den Umgang mit verschiedensten smarten Endgeräten und deren Verwaltung unkompliziert und intuitiv bereitstellt.

Wir entschieden uns dazu, unser Projekt "Gaia" zu nennen, da Gaia die erste griechische Göttin war, die aus dem Chaos hervorgegangen ist. Durch dieses Bild einer allumfassenden Instanz wurde unser Ziel, diverse Smart Home Geräte auf einer einzigen Plattform gebündelt verwalten zu können, zum Ausdruck gebracht.

Entwicklungsumgebung

VSCode

VSCode ist ein Quelltext-Editor der von Microsoft als OpenSource entwickelt wird. Der Editor steht kostenlos zur Verfügung. Auch können verschiedene Entwickler an dem Projekt teilhaben und mitwirken.

Visual Studio Code (VSCode) ist auf allen gängigen Betriebsystemen wie Windows, MacOs und Linux plattformübergreifend verfügbar. Ermöglicht wird das über das Framework Electron. Dieses Framework macht es möglich Webapplikationen als Desktop Apps zu deployen. VSCode ist nämlich in Javascript und Typescript geschrieben und könnte daher in jedem modernen Webbrowser ausgeführt werden.

Es werden nahezu alle gängigen Programmiersprachen unterstützt. Für Syntaxhighlighting, Debugging oder Autovervollständigung stehen zahlreiche Plugins zur Verfügung. Für dieses Projekt waren besonders Plugins wie Angular Snippets, ESLint und GraphQL hilfreich. Für eine schönere Optik der Dateien wurde vscode-icons genutzt. Dieses Plugin ersetzt die Standard Datei Symbole durch zum Dateityp passende.

Um besonders sauberen Code zu schreiben wurde auch CodeMetrics benutzt. Dieses Plugin überprüft die zyklomatische Komplexität von Methoden. Um falsch geschriebene Wörter zu identifizieren wurde CodeSpellChecker benutzt.

Um den Code Style gleich zu halten wurde für das gesamte Projekt ESLint eingeführt. Das sorgt für Code Style Guidelines dafür, dass der Code über das gesamte Projekt konsistent bleibt. Dadurch werden auch viele Merge Konflikte vermieden. ESLint steht für die IDE auch als Plugin zur Verfügung, sodass Missachtungen der Guidelines direkt gehilighted werden können.

IntelliJ IDEA

IntelliJ ist eine IDE des Unternehmens JetBrains und streng genommen für die Programmiersprachen Java, Kotlin, Groove und Scala gedacht. Es ist möglich durch Plugins den Funktionsumfang so erweitern, dass auch Programme in JavaScript bzw. TypeScript problemlos damit entwickelt werden können.

Auch in dieser IDE wurde via Plugin zyklomatische Komplextiät geprüft und ein automatisiertes ESLint durchgeführt.

Projektplanung

Projektidee

Da es mittlerweile viele verschiedene Protokolle und dazugehörige Produkte auf dem Smarthome-Markt gibt soll eine Anwendung

programmiert werden, die sich leicht um neue Übertragungsprotokolle erweitern lässt und diese zu einer einheitliche Schnittstelle zusammenfasst. Zudem soll ein Userinterface geschaffen werden, über das die vorhanden Geräte angezeigt und gesteuert werden können. Es soll dem Anwender möglich sein mit minimalsten technischen Kenntnissen das Produkt einzurichten und smarte Geräte hinzufügen zu können.

Ziel des Projekts

- MQTT als bekanntes Protokoll für unsere Plattform
- Angular Client
- GraphQl im Backend, da minimale Daten und maximal Flexibilität

Grober Projektentwurf

Zu Beginn des Projekts haben wir uns darüber ausgetauscht, wer von uns in welchen Bereichen bereits Erfahrungen sammeln konnte. Da Marcel und Marvin bereits viele Erfahrungen in der Backend-Entwicklung hatten und es sie auch interessierte, arbeiteten sie viel daran. Leia arbeitete sich hingegen in Angular und seine Funktionsweisen ein und widmete sich eher dem Frontend. Für uns war es wichtig, dass sich jeder in seiner Rolle wohlfühlt und Spaß an dem Projekt und der gemeinsamen Realisierung haben wird.

Zum Austausch erstellten wir uns einen kostenlosen Slack-Workspace. Dieser bot uns die Möglichkeit, viele verschiedene Kanäle zu erstellen und in jedem Kanal spezifische projektbezogene Inhalte auszutauschen. Somit fiel es uns auch im Nachhinein leicht, gewünschte Informationen schnell und einfach wiederzufinden. Ein weiterer Vorteil an Slack war es, dass wir auch problemlos Dateien (beispielsweise Bilder oder PDFs) schicken und speichern konnten. Durch diese Funktionen und noch einige mehr war Slack für uns eine gute und unkomplizierte Plattform zum gemeinsamen Austausch.

Zur Übersicht unserer bereits erledigten und bevorstehenden Aufgaben und des darin enthaltenen Codes entschieden wir uns für eine Organisation über GitHub. Für uns war die Planung des Projekts über GitHub bequem und einfach. Durch unsere erstellten Milestones und Tickets konnte man den Stand des Projekts und die Aufgaben, die noch bevorstanden, schnell und ohne Weiteres ablesen. Auch der Austausch des Codes war sehr komfortabel, da er über GitHub und die angebotenen Funktionen stattfand.

Nachdem für uns klar war, wie wir uns austauschen möchten, machten wir uns an unseren groben Projektentwurf. Unser erste grobe Systemarchitektur bestand aus: Client, Embedded MQTT Server und Backend Server.

Umsetzungsübersicht

Während des Projekts arbeiteten wir viel mit Shellys, um unsere Funktionalitäten zu testen. Das hatte den Hintergrund, dass einige von uns bereits solche Geräte im privaten Rahmen besaßen und auch schon viel damit herumexperimentiert hatten. Somit nahmen sie eine wichtige Rolle in unserem Projekt und dessen Testzwecken ein.

Weitere Tests fanden mit einem ESP8266 statt. Diesen verbanden wir mit einem Temperatursensor und entwickelten eine passende Custom Firmware. Genutzt wurde er, um Temperaturen in einem Wohnzimmer zu messen. Unser Ziel war es, diesen ebenfalls mit Gaia zu vernetzen, um die Kompatibilität verschiedener Endgeräte zu erproben.

Um die Daten unserer Plattform zu speichern, entschieden wir uns für eine MongoDB. Dahinter steckte die Überlegung, dass unsere Architekturen bereits sehr JSON-orientiert aufgebaut waren und MongoDB uns daher eine effiziente Integration bot. Für den Datenaustausch vom Client zur API entschieden wir uns für GraphQL. GraphQL stellte uns die Möglichkeit bereit, nur die benötigten Datensätze abzurufen und zu modifizieren, was sich positiv auf unsere Performance auswirkte.

Unsere Plattform läuft über einen Apollo Server. Dieser ist für die Verarbeitung der Daten und der Anbindung zur Datenbank, sowie die Handhabung der Protokollhändler zuständig.

Dank eines Docker Images kann das die gesamte Implementierung einfach auf lokalen Geräten installiert werden. Somit brauchen die Nutzer keine tiefgreifende Kenntnisse in der Verwaltung von Servern, um Gaia zu nutzen.

Zeitplan

Wir haben uns dazu entschieden, während des Projekts mit Scrum zu arbeiten. Für uns schien eine agile Entwicklungsmethode sinnvoller zu sein, da wir uns auf unsere Ideen während des Entwicklungsprozesses einlassen wollten. Es war auf jeden Fall sehr bereichernd flexibel an Gaia zu arbeiten, da uns einige unserer Features während des Entwicklungsprozesses eingefallen sind und wir diese nach unseren Bedürfnissen implementieren konnten.

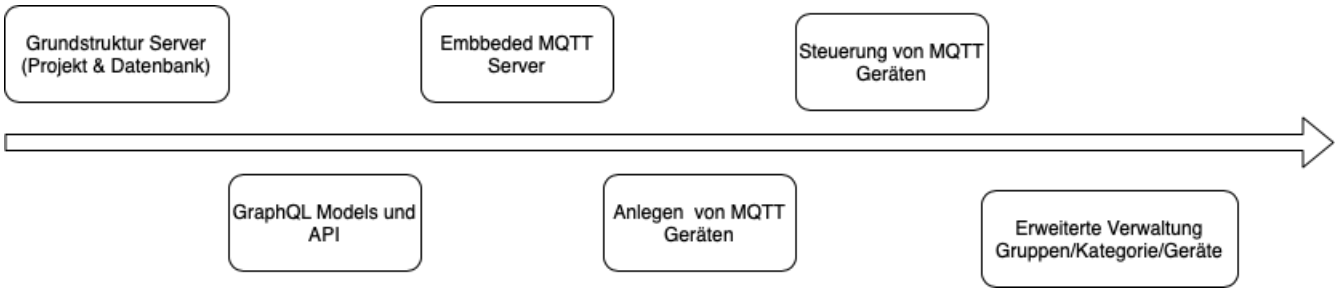
Auf der anderen Seite gab es auch wenige Features bei welchen wir uns vom Aufwand her verschätzt hatten und welche leider nicht so einfach zu realisieren waren wie gedacht. Auch da hat uns die agile Arbeitsweise weitergeholfen, da wir diese Features einfach weglassen oder ersetzen konnten. Unsere Sprints waren meist zwei Wochen lang. Mittwochs haben wir uns im Rahmen der Vorlesung

zu einem Weekly getroffen. Tägliche treffen waren aufgrund anderer Projekte nicht möglich. Sofern möglich haben wir uns virtuell getroffen um aufkommende Fragen beim Programmieren mit kürzeren Wegen abzuklären. Bei komplexeren Aufgaben nutzten wir auch die Möglichkeit des Pair-Programming. Nach jedem Sprintende setzten wir uns neue Ziele, welche wir in dem bevorstehendem Sprint erreichen wollten. Diese Ziele orientierten wir an unseren Meilensteinen (Epics), welche wir uns am Anfang des Projektes zurechtgelegt hatten. Unsere Epics hatten uns vor allem dabei geholfen, eine Orientierung zu haben, welche Punkte uns noch wichtig sind und was uns noch alles bevorsteht. Da Github die Epics nur für Premium Nutzer bereitstellt haben wir die Meilensteine als Epics genutzt.

Wir haben sie wie folgt festgelegt:

Unsere Meilensteine für die **Frontend-Entwicklung**:

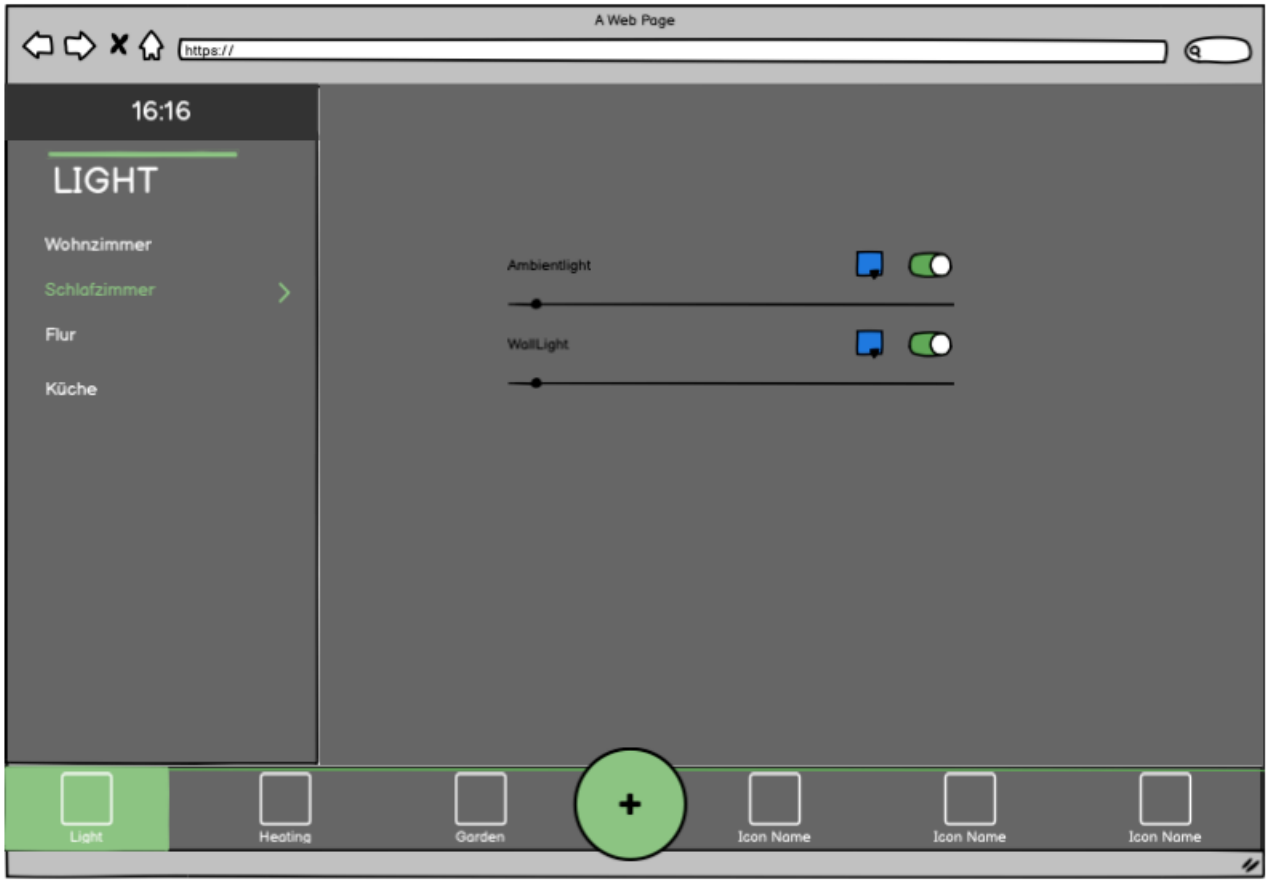
Meilensteine der **Backend-Entwicklung**:



Design

Designfindung

Uns war von Anfang an klar, dass wir unser Design schlicht und übersichtlich halten wollen. Nachdem wir eine Weile über das Design diskutiert hatten, konnten wir unsere Vorstellungen in einem ersten Mockup festhalten:



Da wir unsere Plattform zunächst an der Steuerung von Lichtern getestet hatten, war unser Mockup sehr auf das Thema "Licht" ausgerichtet.

Elemente

Bei einem Blick auf das Mockup werden die verschiedenen Elemente, auf die wir Wert gelegt haben, deutlich:

Obere Leiste

Die obere Leiste sollte alltägliche Informationen wie die Uhrzeit oder auch das Datum auf einen Blick liefern.

Gruppen-Menü

Mit dem Gruppen-Menü (linkes Seitenmenü) wollten wir die Möglichkeit anbieten, selbst erstellte Gruppen von Geräten auf einen Blick zu sehen. Uns war es wichtig, dass die User die Möglichkeit haben, Gaia nach ihren persönlichen Bedürfnissen zu gestalten und zu ordnen.

Content

Auf unserer großen Content-Page sollten die User genügend Platz haben, um die gewünschten Inhalte anzeigen zu lassen. In unseren Überlegungen sollten dort die Widgets wie z.B. ein Widget um Lichter zu steuern, ihren Platz finden.

Widgets:

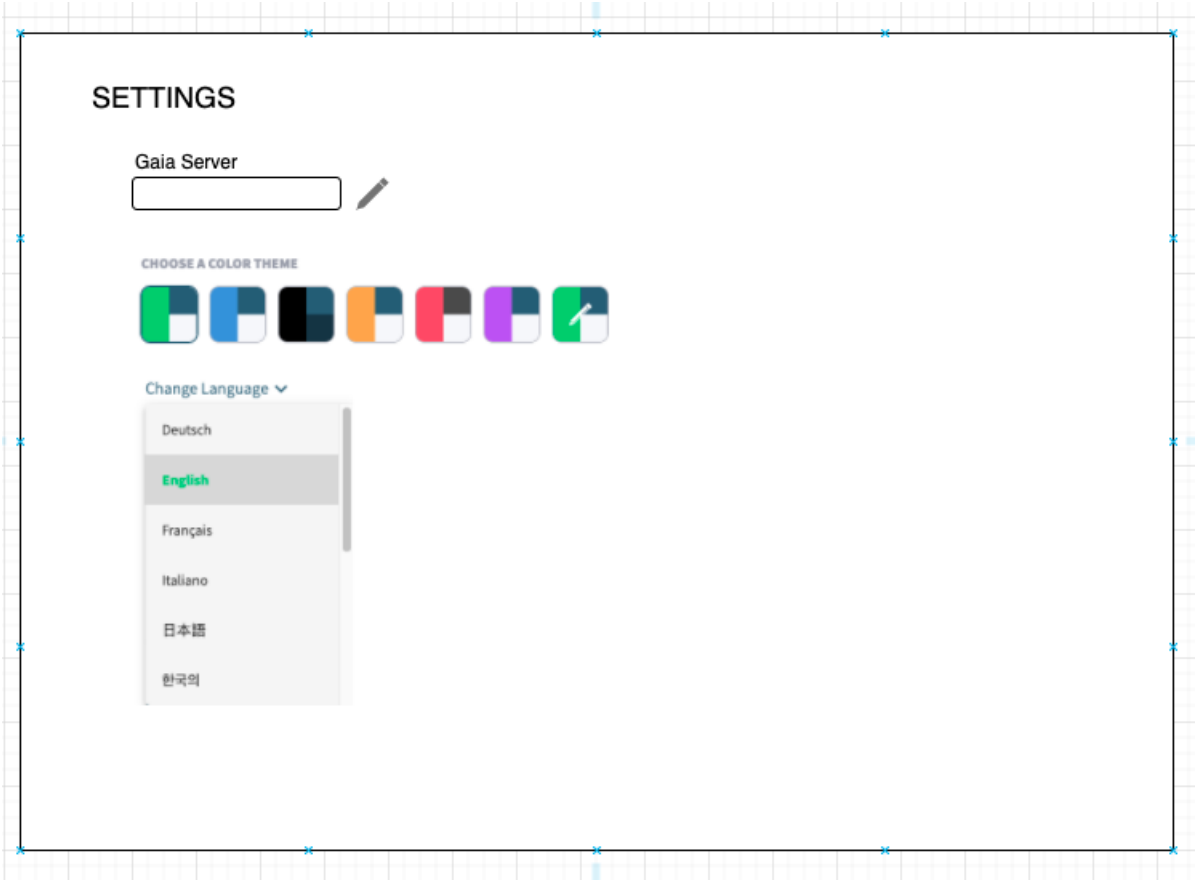
Wir wollten verschiedene Widgets anbieten. Diese sollten individuell auf der Content-Page zusammengestellt werden können. In unseren ersten Überlegungen setzten wir vor allem auf Lichtwidgets den Fokus.

Untere Navigationsleiste

Die untere Navigationsleiste sollte aus zwei wesentlichen Elementen bestehen:

1. **Hinzufügen-Button:** Bei unserer Designfindung wollten wir einen großen Button in der Mitte der Steuerelemente einbauen. Seine Funktion sollte darin bestehen, dass man Geräte einfach und schnell einfügen kann.
2. **Steuerelemente:** Über unsere Steuerelemente sollte man einfach zwischen den verschiedenen Gruppierungen der Endgeräte wechseln können. So kann sich der User beispielsweise alle Geräte in seiner Gruppe "Licht" anzeigen lassen.

Settingspage



Bei der Settingspage war es uns wichtig, dass der Nutzer die wichtigsten Einstellungen auf einen Blick hat. Bei der Eingabe der Server-Adresse war es uns zudem wichtig, dass der User keine versehentlichen Änderungen vornehmen kann. Daher haben wir einen zusätzlichen Editier-Button eingeplant. Änderungen an dem Textfeld können also nur vorgenommen werden, wenn der Editier-Button geklickt wurde.

Des Weiteren fanden wir die Idee, verschiedene Themes anzubieten, schön. Somit sollte die Möglichkeit gegeben sein, die Gaia-App individuell einzurichten.

Außerdem planten wir ebenfalls die Möglichkeit, die gewünschte Sprache einzustellen.

Umsetzung

Angular Material

Um unsere Webapplikation unseren Wünschen gemäß zu gestalten, haben wir uns von Angular Material Gebrauch gemacht. Wir haben viele Module verwendet, welche durch Angular Material bereitgestellt wurden:

- **MatButton**: bietet die Möglichkeit, vielfältig anpassbare Buttons in die UI einzubringen.
- **MatDividerModule**: stellt Teiler zur Verfügung, um die UI übersichtlicher zu gestalten.
- **MatIconModule**: haben wir genutzt, um unsere Applikation mit simplen Symbolen zu verschönern.
- **MatSliderModule**: stellt Drag-and-Drop-Slider zur Verfügung. Diese haben wir vor allem bei unserem Lichtwidget benutzt, um eine Möglichkeit, das Licht zu dimmen, anzubieten.
- **MatSlideToggleModule**: stellt einen simplen Schalter bereit, welchen wir für das an-/ausschalten der Lichter eingebaut haben.
- **MatSidenavModule**: wurde von uns genutzt, um unsere seitliche Navigationsleiste abzugrenzen und zu implementieren.
- **MatCardModule**: bietet die Möglichkeit, kleine "Kärtchen" darzustellen. Diese können mit dem gewünschten Inhalt gefüllt und nach Belieben angezeigt werden. Sie sollten für die Realisierung der Widgets verwendet werden.
- **MatInputModule**: liefert Eingabe-Textfelder. Wurden von uns unter anderem für die Eingabemöglichkeit der Serveradresse genutzt.

- **MatSelectModule**: bringt die Möglichkeit mit sich, ein Auswahlmenü einzubauen. War für uns für die Realisierung der Sprachauswahl in unserem Settingsmenu von Nutzen.

Angular Material bot uns durch die Vielzahl an Modulen eine große Auswahl, um die Vorstellungen unseres Designs umzusetzen. Daher fanden wir es für unser Projekt sehr hilfreich und sehr angenehm damit zu arbeiten. Dies wurde auch durch die Seite von Angular Material unterstützt, da man mit den verschiedenen Designs rumprobieren kann und den Code anschließend bequem zur Verfügung gestellt bekommt.

Flex-Layout

Das Angular FlexLayoutModule haben wir zur Anordnung unserer Elemente genutzt. Somit konnten wir nicht nur den Abstand zu den Rändern oder auch die Ausrichtung der Elemente festlegen, sondern auch entscheiden wie breit und hoch unser Element sein sollte und ob es sich an eine Veränderung der Fenstergröße anpasst.

Raspberry Pi

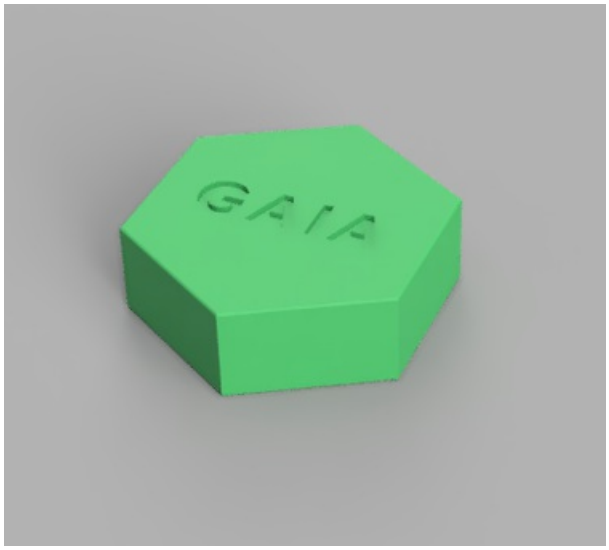
Als Hardware für den Smarthome-Server und Webserver ist die Wahl auf den Raspberry Pi gefallen. Der Raspberry Pi ist ein Einplatinencomputer - ein sogenanntes System-on-a-Chip und mit einer Größe von 85mm x 56mm x 17mm sehr kompakt.

Der Raspberry Pi bietet mit Raspberry Pi OS ein vollwertiges Debian basiertes Betriebssystem und besitzt ein integriertes Wifi-Modul. Die Installation von Docker ist möglich, wodurch wir plattformunabhängig Entwickeln können und im Anschluss die fertige Anwendung unkompliziert auf den Pi portieren können.

Der Pi besitzt eine Vielzahl an I/O Optionen. Diese bieten die Möglichkeit weitere Module anzuschließen. Das ist für die Vision von Gaia als eine Plattform für viele (alle) Smarthome Geräte von Vorteil. Weitere Module wären ein Zigbee Empfänger um auch Geräte im Low Frequenzbereich zu unterstützen ohne das WLAN mit weiteren Geräten zu belasten. Auch ein Array Mikrophone ist denkbar um eine einfache bis aufwendige Steuerung von Geräten über einfache Sprachbefehle zu ermöglichen. Bis hin zur lokalen SmartSpeaker Alternative, welche nur bei Webanfragen auf das Netzwerk zugreift.

Wir arbeiten mit der 3. Generation des Pi's, da wir die Spezifikationen als ausreichend bewerten und es kein Grund gibt ein leistungsstärkeres, aber teureres Model der 4. Generation zu nutzen. Auch die meisten Module sind zwischen der 3. und 4. Generation voll kompatibel. Sofern die Anwendung zu einem späteren Zeitpunkt mehr Rechenleistung benötigt, ist ein Wechsel deshalb sehr einfach. Auch durch das Deployment mit Docker kann die Hardware ohne größeren Aufwand getauscht werden.

Als günstiger und kompakter Einplatinenrechner ist der Raspberry Pi also bestens für das Projekt geeignet. Er kann um viele Module erweitert werden. In einem kleinen Gehäuse untergebracht lässt er sich dezent in den Wohnraum integrieren.



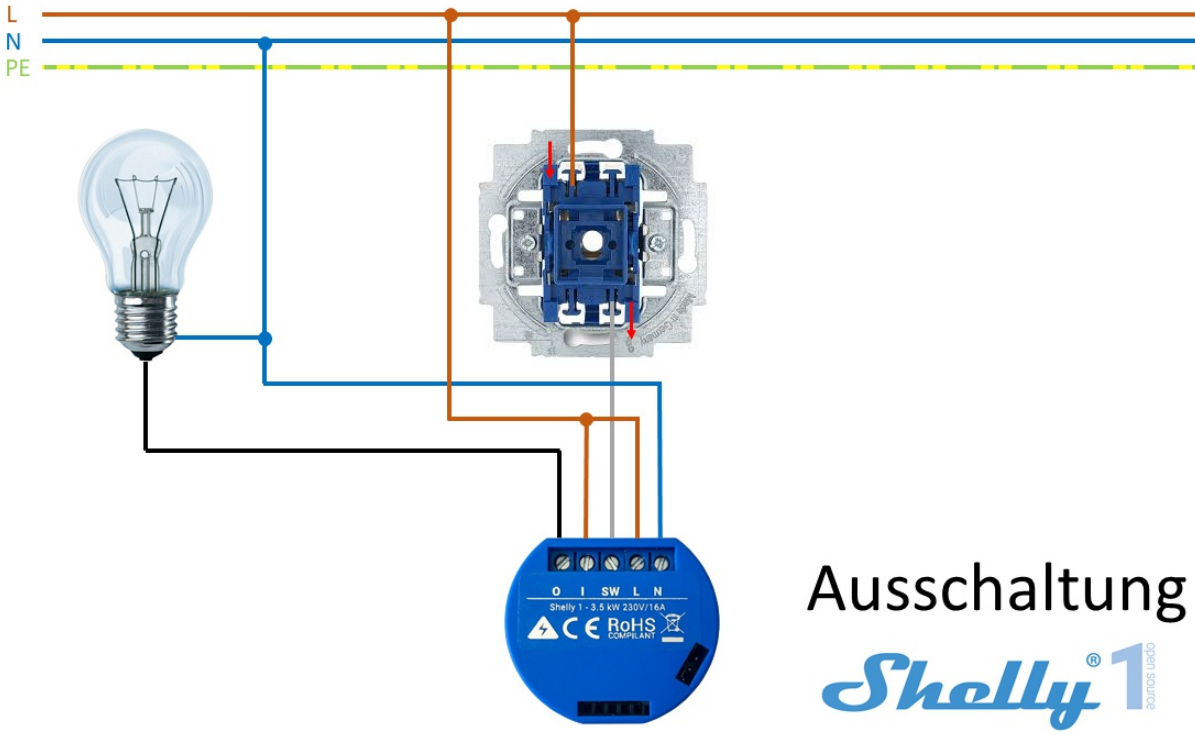
Shelly 1

Als Steuerelemente für das Licht haben wir uns für Shelly 1 entschieden. Das sind WLAN-fähige Unterputzmodule des bulgarischen Herstellers Allterco Robotics. Damit können bestehende Bauten durch Nachrüsten oder auch Neubauten einfach um eine intelligente Steuerung erweitert werden.

Ein wesentlicher Vorteil der Shellies ist dabei, das Lampen weiter regulär mit dem Lichtschalter ausgeschalten, über die Schnittstelle des Shellys aber wieder eingeschaltet werden können. Bei anderen smarten Lösungen wie einer Phillips Hue Glühbirne oder ähnliches, ist die simultane Steuerung nicht möglich.

Ebenfalls für die Shellies spricht der günstige Preis pro Gerät. Über die IO Pins können auch andere Firmwares aufgespielt werden. Um nicht von einer eventuell erzwungenen Cloudanbindung abhängig zu sein, wurden die Shellies mit der OpenSource Firmware Tasmota geflasht.

Die Anwendungsmöglichkeiten von Shellies ist sehr vielseitig. Es können sowohl 230V als auch 12-24V geschaltet werden. So können Lampen, Steckdosen und andere Hausstromgeräte, aber auch Niederspannungsgeräte wie Türöffner oder elektrische Ventile zur Gartenbewässerung gesteuert werden.



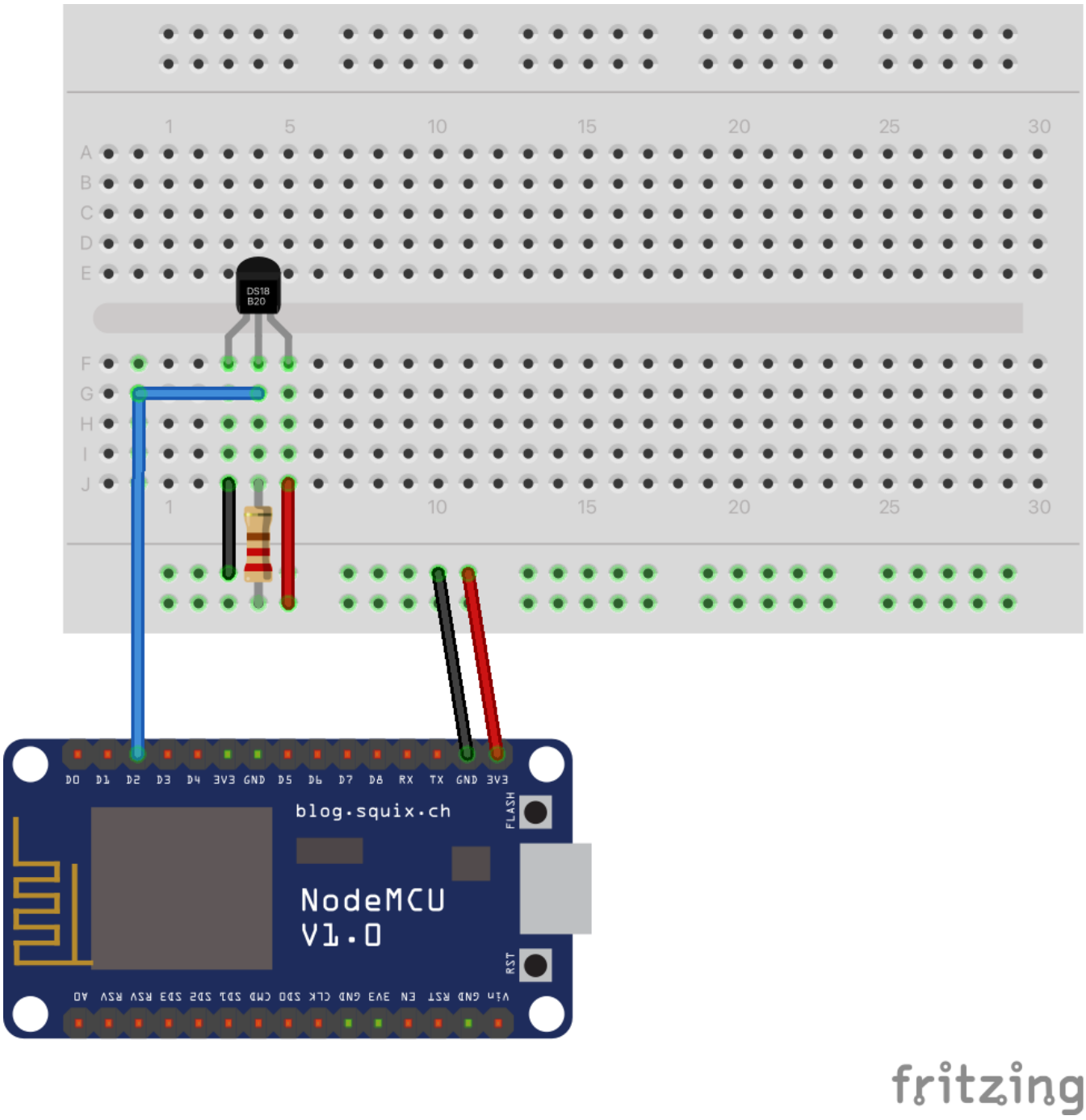
Temperatursenor ESP8266 mit DS18b20

Motivation

Der ESP8266 eignet sich mit seinem integrierten WLAN-Modul ideal um eigene MQTT-fähige Geräte zu entwickeln. Daher haben wir uns dazu entschieden einen Termperatursensor zu entwickeln. Nach kurzer Recherche ist die Entscheidung auf den DS18b20-Sensor gefallen.

Verkabelung

Die Verkabelung ist denkbar einfach. Man benötigt Drei Jumper Kabel, einen 4,7k Ohm Widerstand, einen DS18b20 sowie den ESP8266. Ein Breadboard ist von Vorteil im Entwicklungsprozess. Um den gesamten Aufbau später dann in "Produktion" zu nutzen kann dieser auch auf eine Steckplatine aufgelötet werden.



Sketch für den ESP8266

Der Sketch kann über die Arduino IDE auf den Mikrocontroller aufgespielt werden. Fehlende Bibliotheken können über den Library-Manager der IDE installiert werden.

```
#include <OneWire.h>
#include <DallasTemperature.h>
#include <ESP8266WiFi.h>
#include <Ticker.h>
#include <AsyncMqttClient.h>

//Wifi Credentials
#define WIFI_SSID ""
#define WIFI_PASSWORD ""

// Raspberri Pi Gaia MQTT Broker
#define MQTT_HOST IPAddress(192, 168, 2, XXX)
#define MQTT_PORT 1883 // DEFAULT MQTT Port

// Temperature and Discovery MQTT Topics
#define MQTT_TOPIC_TEMP "gaia/ds18b20/temperature"
#define MQTT_TOPIC_DISCOVERY "gaia/discovery/dsb18b20/config"
```

```
// GPIO where the DS18B20 is connected to
const int oneWireBus = 4;
// Setup a oneWire instance to communicate with any OneWire devices
OneWire oneWire(oneWireBus);
DallasTemperature sensors(&oneWire);

AsyncMqttClient mqttClient;
Ticker mqttReconnectTimer;
WiFiEventHandler wifiConnectHandler;
WiFiEventHandler wifiDisconnectHandler;
Ticker wifiReconnectTimer;


// Temperature value
float temp;
// Time values
unsigned long previousMillis = 0;
const long interval = 10000;

void connectToWifi() {
  Serial.println("Connecting to Wi-Fi...");
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
}

void onWifiConnect(const WiFiEventStationModeGotIP& event) {
  Serial.println("Connected to Wi-Fi.");
  connectToMqtt();
}

void onWifiDisconnect(const WiFiEventStationModeDisconnected& event) {
  Serial.println("Disconnected from Wi-Fi.");
  mqttReconnectTimer.detach();
  wifiReconnectTimer.once(2, connectToWifi);
}

void connectToMqtt() {
  Serial.println("Connecting to MQTT...");
  mqttClient.connect();
}

void onMqttConnect(bool sessionPresent) {
  Serial.println("Connected to MQTT.");
  mqttClient.publish(MQTT_TOPIC_DISCOVERY, 1, true, "{\"topic\":\"gaia/ds18b20/temperature\"}");
}

void onMqttDisconnect(AsyncMqttClientDisconnectReason reason) {
  Serial.println("Disconnected from MQTT.");

  if (WiFi.isConnected()) {
    mqttReconnectTimer.once(2, connectToMqtt);
  }
}

void setup() {
  sensors.begin();
  Serial.begin(115200);
  Serial.println();

  wifiConnectHandler = WiFi.onStationModeGotIP(onWifiConnect);
  wifiDisconnectHandler = WiFi.onStationModeDisconnected(onWifiDisconnect);

  mqttClient.onConnect(onMqttConnect);
  mqttClient.onDisconnect(onMqttDisconnect);
  mqttClient.setServer(MQTT_HOST, MQTT_PORT);

  connectToWifi();
}

void loop() {
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval) {

    previousMillis = currentMillis;
```

```
    sensors.requestTemperatures();
    temp = sensors.getTempCByIndex(0);


    // Publish an MQTT message on topic gaia/ds18b20/temperature
    uint16_t packetIdPub1 = mqttClient.publish(MQTT_TOPIC_TEMP, 1, true, String(temp).c_str());

    Serial.printf("Publishing on topic %s, packetId: %i ", MQTT_TOPIC_TEMP, packetIdPub1);
    Serial.printf("Message: %.2f \n", temp);
  }
}
```

Angular

Angular ist ein TypeScript-basiertes Framework zur Entwicklung von Front-End-Webapplikationen.

Wir haben uns dazu entschieden mit Angular zu arbeiten, da einige Gruppenmitglieder bereits Erfahrungen mit Angular sammeln konnten und die Idee des MVVM (Model View ViewModel) Entwurfsmusters von Angular sehr gut fanden. Uns gefiel besonders, dass sich Controller und View durchgehend parallel synchronisieren.

Angular bietet umfangreiche Möglichkeiten, um geschriebenen Code einheitlich zu gestalten. Das hat uns in diesem Projekt sehr dabei geholfen, unseren Code homogen zu halten. Dadurch war es einfacher, sich in die geschriebenen Zeilen einzufinden und daran arbeiten zu können. Diese Gefühl der Konsistenz sorgte dafür, dass das Coding-Erlebnis für uns als sehr angenehm wahrgenommen wurde. Auch um den Aufbau der Komponenten und Services mussten wir uns nicht so viele Gedanken machen. Der Grund dafür war, dass es sehr identische "Baupläne" gab und wir uns recht schnell eingefunden hatten.

Ein weiterer Punkt war für uns der Aspekt der Wartbarkeit. Der Grund dafür war, dass wir uns auch in Zukunft noch mit dem Projekt auseinandersetzen wollten. Da Google innerhalb des Unternehmens selbst Angular bei der Entwicklung von Anwendungen benutzt, konnten wir davon ausgehen, dass Angular weiterhin regelmäßig gewartet und weiterentwickelt wird. Für die Überlegungen unserer zukünftigen Pläne, spielte dieser Aspekt daher ebenfalls eine Rolle.

Was uns persönlich ebenfalls an Angular gefiel war die Organisation des Codes. Durch das Prinzip, Code in einem oder mehreren Modulen zu ordnen, gewann unser Projekt unheimlich viel an Übersichtlichkeit. So haben wir uns jederzeit schnell im Code, und dessen Struktur, zurechtgefunden. Auch die Möglichkeiten zur frühzeitigen Fehlererkennung sind uns bei Angular immens ins Auge gestochen. Da Angular auf TypeScript basiert, können wir Typischer entwicklen und Fehler vorab vermeiden. Die Möglichkeit, den Code während der Laufzeit direkt im Browser zu debuggen, war von uns eine der Funktionen, die von uns während der Entwicklung, und vor allem während der Fehlersuche, intensiv genutzt wurde. Oft haben wir die Fehler in unserer Programmierung nicht sofort gefunden und mithilfe dieses nützlichen Features war es dennoch möglich, das Problem ausfindig zu machen.

Eine große Rolle bei der Entscheidung war zudem, dass sowohl im Frontend als auch im Backend in diesem Projekt basierend auf Node.js und TypeScript gearbeitet wurde und somit die Lernkurve deutlich steiler ausfiel.

Zur Einarbeitung und Auffrischung der Kenntnisse empfanden wir den Online Kurs auf Pluralsight als sehr gute Basis, um dem Projekt gerecht zu werden.

Docker

Docker ist eine Plattform zur Prozess-Virtualisierung mithilfe von Containern. Sie erleichtert die Entwicklung, das Testen sowie den Betrieb der gesamten Anwendung erheblich, weil hier Abhängigkeiten auf Bibliotheken und Versionen des Betriebssystems vermieden werden. Sämtliche Bibliotheken/Abhängigkeiten, die von der Anwendung zum Betrieb benötigt werden, sind in einem sogenannten Image gebündelt.

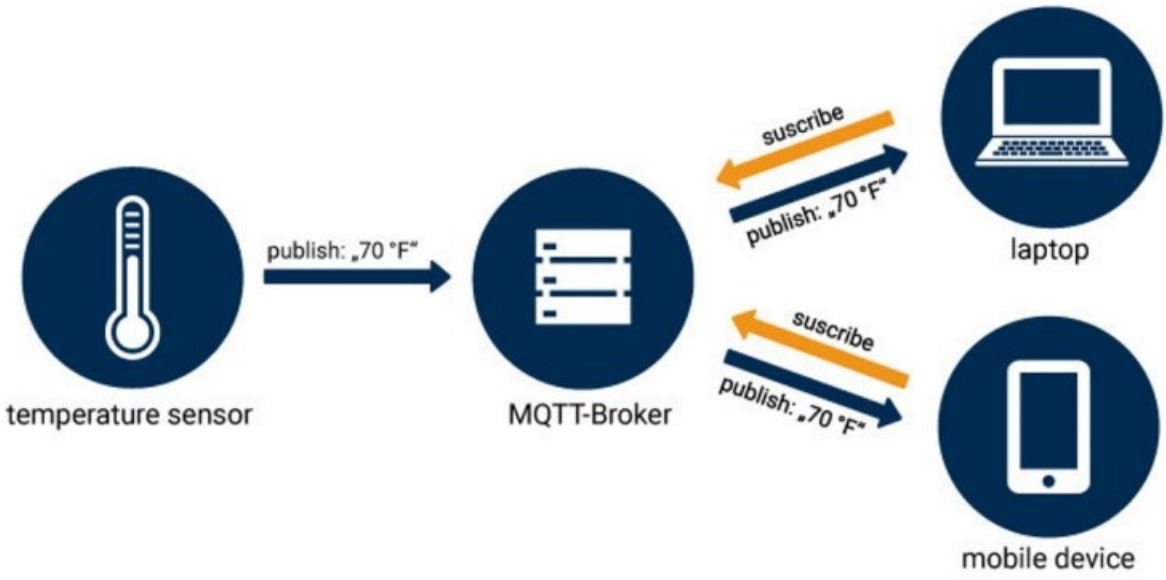
Aus diesem Image kann dann ein lauffähiger Container erstellt werden. Diese Container enthalten selbst keine eigenes Betriebssystem sondern stützen sich auf den Kernel des Host-Betriebssystems. Daher sind Container Anwendungen im Vergleich zu VMs ressourcenschonender. Zudem ist die Bereitstellung denkbar einfach. Von einer Registry (wir nutzen Docker Hub) kann das fertige Image gepullt und zumeist mit einem Einzeiler gestartet werden.

MQTT

Message Queuing Telemetry Transport (MQTT) ist ein offenes Netzwerkprotokoll für die Kommunikation zwischen Maschinen. Wir nutzten es, um Daten in Form von Nachrichten zwischen unserem Hub und unseren smarten Endgeräten auszutauschen.

MQTT ist sehr weit verbreitet, da es eine recht geringe Komplexität aufweist. Auch für uns war das ein Grund, welcher für die Nutzung von MQTT sprach, dass unsere Testgeräte (Shellys) MQTT fähig waren und es somit eine angenehme Lösung für uns darstellte. Auch die relativ geringe Datenmenge bei Übertragungen kam uns zugute. Der Grund hierfür ist, dass das Protokoll sehr schlank ist.

Generell kann man sagen, dass MQTT recht simpel gehalten wurde. Somit ist es perfekt geeignet für Einplatinencomputer und Mikrocontroller. Des Weiteren erlaubt MQTT die beidseitige Kommunikation zwischen Gerät und Hub, was bei der gruppenweisen Ansteuerung und Abstimmung der Geräte förderlich war.



MongoDB

MongoDB ist ein dokumentenbasiertes Datenbanksystem. Somit muss man keine SQL-Abfragen generieren, sondern hat alle Daten ganz bequem in einem Dokument gespeichert. Das hat Vorteile in der Performance, da die Daten nicht erst aus mehreren Tabellen zusammengeführt werden müssen. Auch die Flexibilität und Skalierbarkeit profitieren davon. Die Infrastruktur kann vergleichsweise einfach an die Anforderungen angepasst werden.

Für uns ist die Wahl auf die MongoDB gefallen, da wir viel mit JSON-ähnlichen Dokumenten gearbeitet haben. Diese stellt für uns eine gute Möglichkeit, um unsere Daten zu verwalten. Auch wenn wir im Nachhinein etwas an unseren Datenmodell ändern wollen, können wir dies mithilfe der Mongo Datenbank schnell und unkompliziert erledigen. Außerdem erlaubt sie uns einen einfachen Zugriff auf verschachtelte Daten ohne komplizierte Workarounds.

Ein weiterer Punkt, der für die MongoDB spricht, bestand darin, dass sie in unseren Technologiestack effizient zu integrieren war.

Tasmota

Als Firmware für die Shellies haben wir uns für Tasmota entschieden. Eine Open-Source Firmware, welche ursprünglich nur für Sonoff Geräte entwickelt, wurde. In der Weiterentwicklung wurde die Firmware für Geräte die auf ESP8266 Mikrokontroller basieren erweitert.

Der grundlegende Gedanke hinter Tasmota ist, Geräte von undurchsichtiger Firmware zu befreien und in System einzusetzen, in denen diese nie so vorgesehen waren. Als quelloffenes Produkt kann jeder die genaue Funktionen der Firmware nachvollziehen und sich an der Weiterentwicklung beteiligen. Inzwischen unterstützt Tasmota viele Geräte und Funktionen. Es können Schalter, Lampen oder auch Sensoren mit Tasmota geflasht werden.

Grundsätzlich sind zwei Arten von Geräten für Tasmota geeignet. Das sind auf der einen Seite Sensoren und auf der anderen Seite Akteure. Mit Tasmota als installierter Firmware können Geräte den Status von Knöpfen und Schaltern, sowie viele Arten von Sensoren wie Temperatur, Feuchtigkeit, Helligkeit usw. lesen. Es können Geräte wie Relais, LEDs, IR transmitter und viele mehr geschalten werden.

Tasmota nutzt unter anderem MQTT als Übertragungsprotokoll und bietet mit einer simplen Weboberfläche Möglichkeiten zur Konfiguration. Da Gaia auf lokale Lösungen baut ist Tasmota als Firmware für Geräte besonders gut geeignet. Um die Firmware zu nutzen, ist keine Cloudanbindung notwendig. Alles kann lokal über Wifi im 2,4 GHz oder 5 GHz Netz betrieben werden.

Tasmota ist somit die beste Möglichkeit um ESP8266 basierte Boards in ein vielseitiges IoT Gerät zu verwandeln.

Client

Bereits im Frontend ist es wichtig, die Struktur des Projekts möglichst klar und einfach zu halten. Da das gesamte Projekt auf eine gute Skalierbarkeit abzielt, ist es wichtig, das auch der Client auf diese Änderungen schnell und einfach angepasst werden kann. Die Komponenten bezogene Architektur von Angular bietet daher wesentliche Vorteile für das Projekt.

Für einen Client mit dem Angular Framework, sind die Komponenten die wichtigsten Bausteine. Bei Gaia sind diese für das anzeigen

und interagieren mit der UI zuständig. Hier werden User Events verarbeitet und es erfolgt ein unidirektionaler von den Parent zu den Child Komponenten.

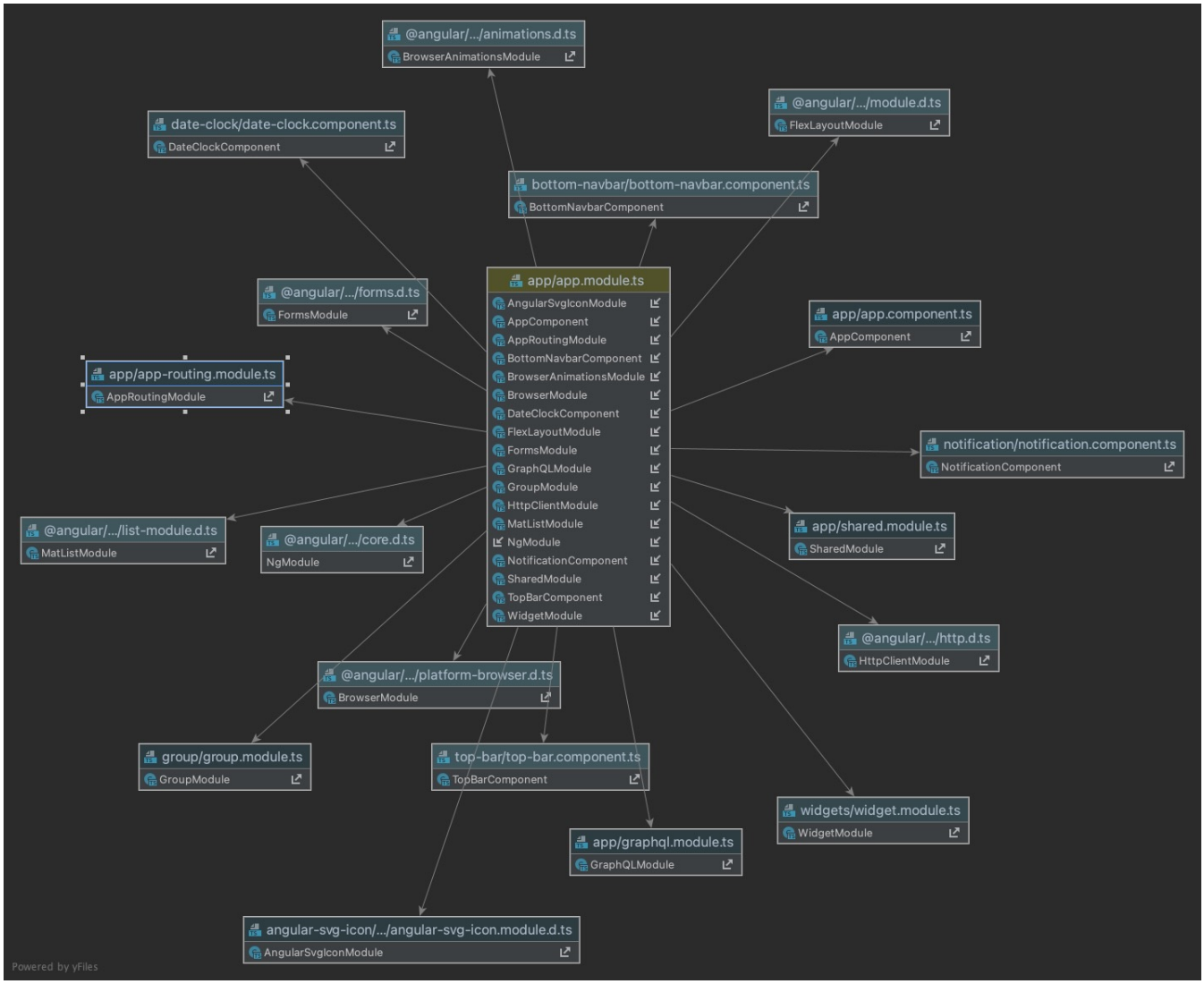
Die Komponenten sind über das Child-Parent Prinzip verbunden, das bedeutet im Parent werden über @Output Daten and das Child weitergeben, welches über @Input die Daten annimmt. Möchte das Child Daten verteilen, werden Events gestartet, auf die jede Komponente, also auch der Parent, reagieren kann.

Die verschiedenen Seiten des Frontends werden über Angular Routen gesteuert. Über Angular Routing können Komponenten dynamisch geladen und angezeigt werden. Das Routing ist eines der wesentlichen Bestandteile um das Frontend als Single Page Applikation bereitzustellen. Da Gaia eine Plattform für kleine SmartHomes bis zur kompletten GebäudeAutomation werden soll, sind die Routen direkt mit LazyLoading eingebunden. Die Geräte der Gruppen beispielsweise, werden erst dann geladen, sobald die Gruppen auch auf der UI angezeigt werden. Dadurch wird die Payload beim aufrufen der Applikation stark reduziert und die Performance verbessert.

Die Daten für die App werden entkoppelt von den Komponenten über Services bereitgestellt. Über Dependency Injection greifen die Komponenten auf die Services zu und starten das abholen der Daten vom Server. Die Services nutzen den Apollo Client um mit GraphQL Anfragen and die API zu schicken. Die Queries und Mutations dazu werden in GraphQL Dateien angelegt.

Durch ein externes Tool werden automatisch Typen vom Server und die Queries und Mutations vom Client zu einer Datei generiert, von der aus eben diese importiert und genutzt werden können. Das sorgt für eine Typsicherheit am Client und eine einfachere Nutzbarkeit.

Die ganze Struktur ist Angular typisch als Baum aufgebaut. Das bedeutet es gibt einen Einstiegspunkt in die Applikation von der aus alle Komponenten voneinander Erben.



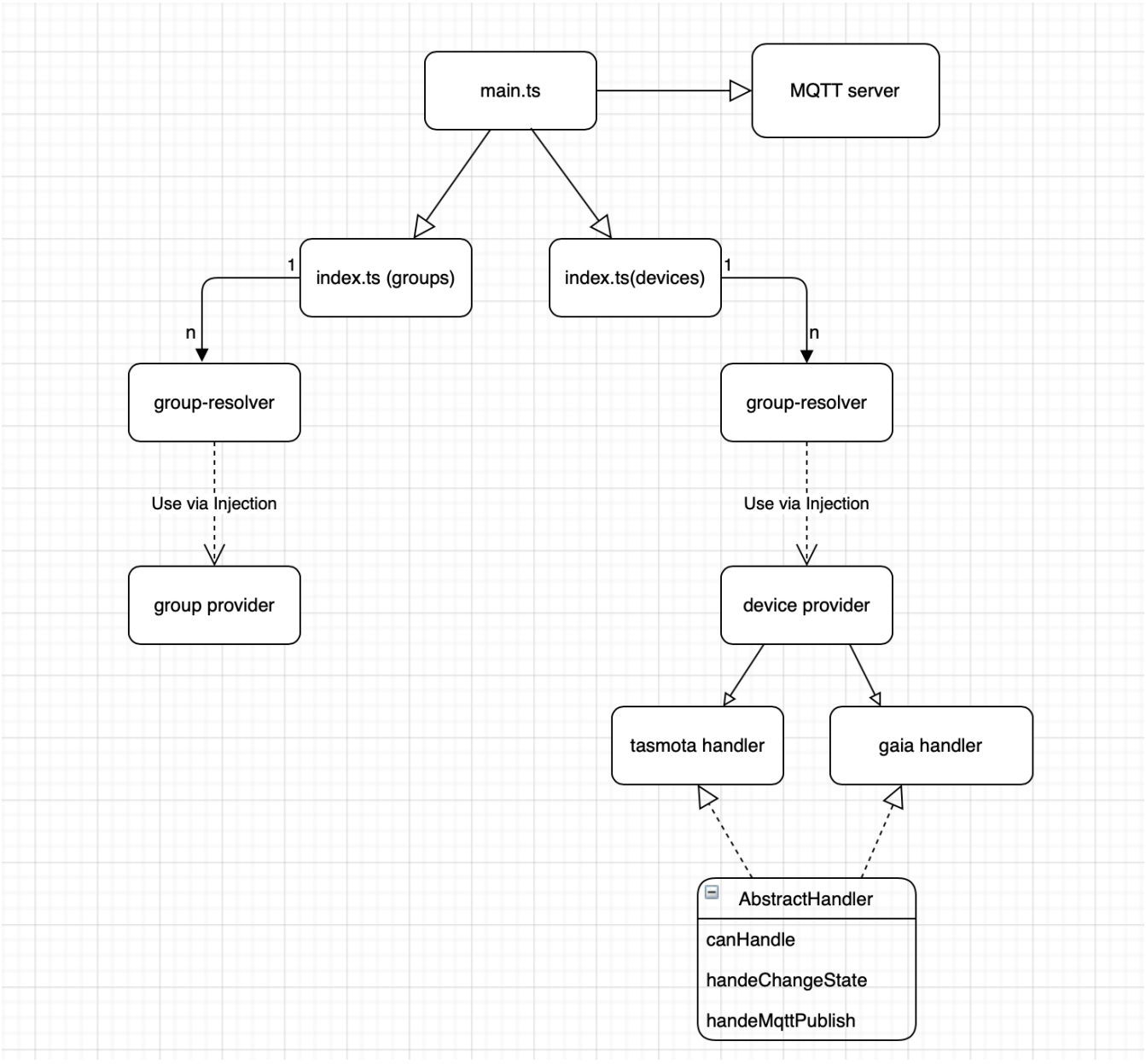
Server

Das Herzstück unserer Anwendung ist der Server. Unser Server basiert auf Apollo Server im Kombination mit Express.

Das Ziel war es die Architektur so zu gestalten, das zukünftige Weiterentwicklungen einfach integriert werden können ohne die Basis verändern zu müssen. Hierbei setzen wir auf die Graph-QL Module Bibliothek. Aktuell gibt es die zwei Module `Device` und `Group`. Ein einzelnes Modul enthält ein Schema, zugehörige Resolver sowie ein Provider für die Datenbankanbindung. Der Provider wird an den benötigten Stelle via `Dependancy Injection` injiziert.

UML

Das Diagramm zeigt schematisch den Aufbau.



Embedded MQTT Broker

Um mit MQTT Servern kommunizieren zu können, wird parallel zum Apollo Server noch ein MQTT Broker benötigt. Wir nutzen den Aedes MQTT Broker. Dieser leitet sämtlichen Traffic über eine GraphQL Mutation weiter. Somit ist die Business Logik gänzlich getrennt vom MQTT Broker. Durch diese Aufteilung ist es auch möglich, einen externen MQTT Broker zu nutzen.

Automatisiertes Anlegen von Geräten

Ein definiertes Ziel war es, das Anlegen von Geräten so einfach wie möglich zu gestalten. Daher wurde eine Möglichkeit geschaffen, die minimalsten Einsatz vom Anwender erfordert. Bei der Einrichtung der Tasmota-IoT-Geräte muss zusätzlich die IP-Adresse des MQTT Brokers sowie der Port (Standard 1883) angegeben werden. Damit ist die Konfiguration abgeschlossen und schon ist das Gerät **Gaia-Ready**.

Tasmota

Tasmota veröffentlich bei der Anmeldung am Broker einmalig eine Konfiguration. Diese enthält alle notwendigen Informationen, um das Gerät in Gaia anlegen zu können. Im Folgende schauen wir uns eine Konfiguration genauer an. Diese ist auf die notwendigen

Eigenschaften gekürzt, die wir verarbeiten.

```
{
   "dn":"Ecklicht",
   "t":"tasmota_644419",
   "rl":[
      1,
      0,
      0,
      0,
      0,
      0,
      0,
      0
   ]
}
```

- "dn" steht für DeviceName und enthält den Namen, den das Gerät bei der Konfiguration erhalten hat.
- "t" steht für Topic. Dieses Topic ist notwendig, um gezielt dieses Gerät anzusprechen.
- "rl" steht für Relais. Tasmota unterstützt Geräte mit bis zu 8 integrierten Relais, die sich unabhängig voneinander schalten lassen. Die Beispielhafte Konfiguration gibt an, dass nur Relais 1 aktiv ist.

Aus diesen Werten erzeugen wir ein `Device` Objekt in unserer Datenbank.

Generalisierung

Das beschriebene Vorgehen ist stark abhängig vom Tasmota Discovery Publish. Um zukünftig auch Firmware von anderen Herstellern unterstützen zu können, ist die Verarbeitung mit dem Chain-of-Responsibility-Pattern gelöst. Das bedeutet, es gibt für jede Firmware einen Handler der von der Abstrakten Klasse `ÀbstractHandler` erbt. Der Device-Provider hält eine Liste mit allen verfügbaren Firmware Handlern.

```
abstract class AbstractHandler {
  abstract canHandle(data: String): boolean; #1

  abstract handleChangeState(device: Device, fn: string, state: string, dataBaseConnection: any); #2

  abstract handleMqttPublish(topic: any, payload: any, dataBaseConnection: any); #3
}
```

1. Diese Methode wird genutzt um herauszufinden, ob der Handler mit dem aktuellen Payload umgehen kann.
2. Verarbeitung von Payload gesendet vom Client
3. Verarbeitung von MQTT Payload

Ein Payload wird abgearbeitet, indem jeder verfügbare Handler "gefragt" wird, ob er diesen Payload verarbeiten kann. Wenn dies zutrifft, wird der Payload je nach Ursprung verarbeitet. Im Code sieht diese Prozedur wir folgt aus:

```
if (handler.canHandle(topic)) {
 handler.handleMqttPublish(topic, payload, mongoDevices);
}
```

Aktuell können wir sowohl Geräte mit Tasmota sowie unsere eigene Gaia Firmware verwalten.

Datenmodell

Unser Datenmodel besteht aus 2 Kollektionen. Zum einem gibt es eine List mit den Gruppen. Hier wird lediglich der Name mit einer Id ($oid: Object Identifier) eingespeichert.

```
{
   "_id":{
      "$oid":"60bcd97aceea0a6c6f631526"
   },
   "name":"Wohnzimmer"
}
```

Unsere zweite Kollektion Beinhaltet die Geräte. Aktuell gibt es ausschließlich Datenmodele für MQTT. Es ist aber möglich auch zukünftig weitere Modelle für andere Protokolle in diese Kollektion einzubringen. Unter "fn" sind unsere möglichen Funktionen des jeweiligen Geräts gespeichert. Das Beispiel zeigt eine Konfiguration für ein Relais, also einen Schalter mit den zwei Zuständen AN oder AUS. Hier könnten auch weitere Relais sowie ihren Zustand angelegt und gespeichert werden.

```
{
   "_id":{
      "$oid":"60a26a765f91ff38053d2baf"
   },
   "name":"Büroooooo Licht",
   "gateway":"TASMOTA",
   "mqttTopic":"tasmota_C490CF",
   "fn":{
      "power":{
         "relay1":"ON"
      }
   },
   "groups":[
      {
         "$oid":"60a50866182b4095787b97b1"
      },
   ],
   "category":"Light"
}
```

Unser Sensor beinhaltet unter "fn" andere Werte:

```
   "fn":{
      "sensor":{
         "sensor1":"23.00"
      }
```

Hier gibt es ausschließlich einen Senorwert, der aktualisiert wird. Zudem wäre es auch möglich ein Gerät mit Relais und Sensor anzulegen, also eine Kombination aus den beiden Beispielen.
Die Funktionen können beliebig mit zukünftigen Entwicklungen des Backendes erweitert werden um auch andere noch nicht bekannte Geräte hinzuzufügen.

Deployment

Wir haben uns dazu entschieden unsere Anwendung als Docker Images zur Verfügung zu stellen. Der Vorteil darin liegt in der einfachen Einrichtung bzw. der geringen Voraussetzungen. Der Nutzer benötigt lediglich eine Docker Installation.

Wie wird aus der Anwendung ein Image?

Die Grundlage zur Dockerisierung der Anwendung ist das sogenannte Dockerfile. Das Dockerfile definiert welche Software zusätzlich zur Anwendung installiert werden muss und gibt an mit welchem Befehl die Anwendung startet. Im Beispiel genauer das Dockerfile des Servers erläutert:

```
# Als Basis dient ein leichtgewichtiges Image mit vorinstallierten Node.js 12.x
FROM node:12-alpine

# WORKDIR erstellt ein Arbeitsverzeichnis im Image
WORKDIR /usr/src/app

# Kopieren der package*.json um notwendige Abhänigtkeiten  zu installieren
COPY package*.json ./
RUN npm ci

COPY . .

# Explizite Angabe des freigelegten Ports
EXPOSE 4000

# Befehl mit dem die Anwendung gestartet wird
ENTRYPOINT [ "npm", "run", "prodServer" ]
```

Das Dockerfile für gaia-ui enthält zudem den Befehl zu Installation eines NGNIX. Dieser Webserver ist zuständig für die Auslieferung der Website.

```
FROM node:12.16.1-alpine As builder

WORKDIR /usr/src/app

COPY package.json package-lock.json ./

RUN npm ci

COPY . .

// Befehl zum Build der Angular Application
RUN npm run build --prod

Installation von nginx
FROM nginx:1.15.8-alpine

COPY --from=builder /usr/src/app/dist/gaia-app/ /usr/share/nginx/html
```

GitHub Actions

Workflow

Der Workflow deployt auf einen Pull Request auf main eine neue Version des Images zu Dockerhub. Sowohl für gaia-server sowie gaia-ui wird dieser Workflow ausgeführt. Folgender Ablauf beschreibt die Funktion des Workflows:

1. Das Repository wird auf dem Runner geclont
2. Der Runner meldet sich mit den vorgegebenen Anmeldedaten bei Dockerhub an
3. Die Github Action buildx baut die Anwendung für verschiedene Plattformen (linux/amd64, linux/armv7, linux/arm64)
4. Unter dem angegebenen Tag wird das fertige Image veröffentlicht

```
name: Docker_Image_Dockerhub_CD

on:
  pull_request:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Docker login
        env:
          DOCKERHUB_USERNAME: ${{ secrets.DOCKERHUB_USERNAME }}
          DOCKERHUB_PASSWORD: ${{ secrets.DOCKERHUB_PASSWORD }}
        run: |
          docker login -u $DOCKERHUB_USERNAME -p $DOCKERHUB_PASSWORD
      - name: Install buildx
        id: buildx
        uses: crazy-max/ghaction-docker-buildx@v1
        with:
          buildx-version: latest
      - name: build the Docker image and push to dockerhub
        run: |
          docker buildx build --push \
          --tag pigtastic/gaia-server:latest \
          --platform linux/amd64,linux/arm/v7,linux/arm64 .
```

Dockerhub

Die fertigen Images können von Dockerhub bezogen werden.

Installation

Vorraussetzungen

Es wird eine aktuelle Docker Installation sowie `docker-compose` > 1.28.0 benötigt.

Ablauf

Um das gesamte Projekt zu starten, muss in das Starter-Verzeichniss gewechselt und das passende Skript ausgeführt werden. Aktuell unterstützen die Skripte die Installation auf Linux sowie im speziellen für den Raspberry Pi.

Das Skript Pi sieht wie folgt aus:

```
#!/bin/sh
echo 'Start...\n'
[...]
# Get Docker Images
echo 'Pull docker images\n'
docker pull pigtastic/gaia-ui
docker pull pigtastic/gaia-server
docker pull andresvidal/rpi3-mongodb3

echor "Create mongodb directory"
 mkdir -p ~/mongo/data

# Start Containers
echo '\nStart-up\n'
docker-compose -p gaia-home --profile pi up --detach

echo '\n FINISH'
```

Die Gaia-Images werden von Dockerhub geladen, zudem wird ein Image von MongoDB passend für den Raspberry Pi gepullt.
Als nächstes wird mit `docker-compose -p gaia-home --profile pi up --detach` der Verbund aus Container gebaut und im Hintergrund gestartet. Über die Option `--profile` werden die korrekten Services für den Raspberry Pi genutzt. Als Optionen stehen

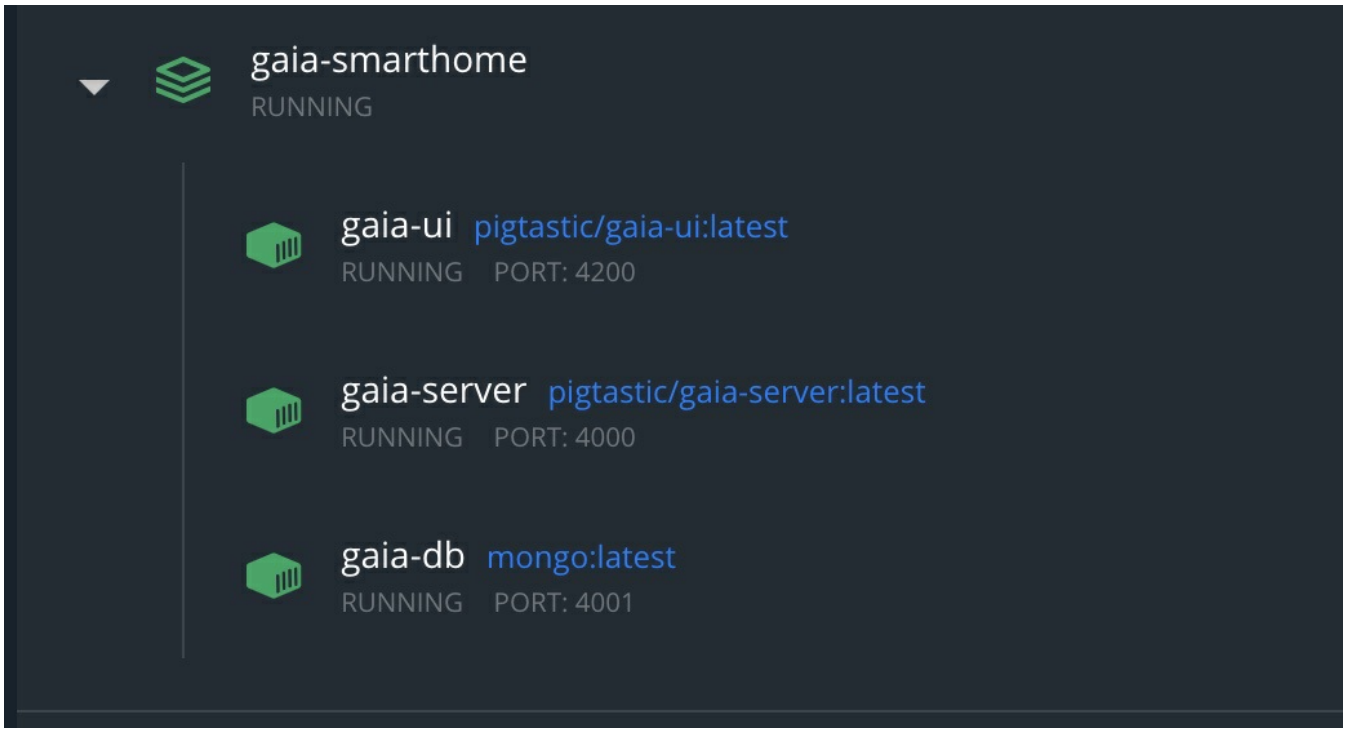
"pi" oder "linux" zur Wahl. Da wir das docker-compose.yaml mit einem Profil nutzen, können wir mit einer Datei alle Betriebssystem abdecken. Sofern in Zukunft weitere Betriebssysteme unterstützt werden, kann die Datei entsprechend erweitert werden.

Die docker-compose.yaml Datei sieht wie folgt aus:

```
version: '3'

services:
  userinterface:
    image: pigtastic/gaia-ui:latest
    container_name: gaia-ui
    restart: always
    ports:
      - "4200:80"
    networks:
      - gaia-network

  server:
    image: pigtastic/gaia-server:latest
    container_name: gaia-server
    restart: always
    ports:
      - "4000:4000"
      - "1883:1883"
    networks:
      - gaia-network

  mongodb:
    image: mongo:latest
    profiles: [ "linux" ]
    container_name: gaia-db
    restart: always
    ports:
      - "4001:27017"
    volumes:
      - ~/mongo/data:/data/db
    networks:
      - gaia-network

  mongodb-pi:
    image: andresvidal/rpi3-mongodb3:latest
    profiles: [ "pi" ]
    restart: always
    ports:
      - "4001:27017"
    volumes:
      - ~/mongo/data:/data/db
    networks:
      - gaia-network

networks:
  gaia-network:
    name: gaia-network
```

Das Compose File definiert wie die einzelnen Images miteinander "verschnürt" werden. Um gegenseitig auf die offenen Ports zugreifen zu können werden alle Services einem gemeinsamen Netzwerk zugewiesen. Die Tags `ports` verknüpfen die Ports der Container mit den Ports des lokalen Geräts, damit die Client Website aufgerufen werden kann. Der Server sowie die Datenbank müssen ihre Ports nicht veröffentlichen. Um besser am Produkt entwickeln zu können ist dies aber dennoch der Fall.
Die folgende Abbildung zeigt die Terminal-Ausgaben während der Installation.

Anschließend läuft das gesamte Produkt im Hintergrund und kann z.B. über die Docker Desktop App überwacht und gesteuert werden. Hier ist es auch möglich über das Terminal auf dem jeweiligen Container Befehle auszuführen oder einfach nur Log-Meldungen einzusehen. Die folgende Abbildung zeigt das fertige Produkt.



Erreichtes Ziel

Im Rahmen der Findungsphase hatten wir uns viele Gedanken gemacht, was unsere Plattform können soll. Bei einem Blick auf unsere Projektplanung lässt sich sagen, dass wir unsere Idee wie gewünscht umsetzen konnten. Wir wollten Gaia entwickeln, da nicht alle smarten Endgeräte mit jeder Plattform kompatibel sind und uns das in der Vergangenheit bereits mehrmals vor Probleme gestellt hat. Darüber hinaus haben sich auch während der Entwicklung neue Ideen für Features ergeben, welche wir ebenfalls implementiert haben. Es gab jedoch leider auch Features die wir uns zwar im Vorneherein gewünscht haben, allerdings hat die Zeit nicht mehr gereicht oder es war doch mit aufwändigeren Workarounds verbunden, diese Features umzusetzen.

Unsere Funktionalitäten

Basis

Server

Beim Server war besonders die Skalierbarkeit wichtig. Da der Server einfach für weitere Protokolle und Geräte erweiterbar sein sollte war hier viel Arbeit nötig. Mit der Zuständigkeitsverkettung der aufkommenden Anfragen zu den jeweiligen Handlern kann der Server mit dem Hinzufügen von Handlern einfach um andere Protokolle und Firmwares erweitert werden.

Durch die Umsetzung der API mit GraphQL können auch hier einfach neue Typen wie Rolladen, Fensterkontakte oder Heizungssteuerungen integriert werden. Die Anfragen können für diese Typen leicht erweitert werden. Insgesamt erhält der Server dadurch eine gute Wartbarkeit.

Webapplikation + App

Eine unserer grundlegenden Funktionalitäten, ohne welche unsere Projektidee nicht umsetzbar gewesen wäre, stellt die Webapplikation dar. Unsere Webapplikation bietet das Benutzerinterface. Da wir eine einfache und übersichtliche Plattform anbieten wollten ist sie also besonders wichtig. Daher haben wir uns viele Gedanken über das Design und die Steuerelemente gemacht, um unser Ziel einer anwenderfreundlichen Plattform zu erfüllen. Darüber hinaus haben wir eines unserer optionalen Features realisiert: eine dazugehörige App. Die App war für uns insofern wichtig, dass wir auch die Möglichkeit bieten wollten, Gaia von überall steuern zu können.

Geräte hinzufügen

Ebenfalls eine grundlegende Funktion stellt das Hinzufügen von Geräten dar. Denn was bringt uns eine Plattform um smarte Einrichtung zu steuern, wenn keine Geräte hinzugefügt werden können? Das Einpflegen von MQTT Geräten übernimmt Gaia beinahe vollautomatisch. Sobald die MQTT-Schnittstelle des Endgeräts aktiviert wurde, erkennt Gaia es und zeigt es an. Nun kann man ganz bequem das Gerät in die Liste einfügen und nach Belieben benennen.

Geräte ansteuern

Die Ansteuerung der Geräte und die Änderung des Status sind ebenfalls elementare Funktionen. Somit können wir mit Gaia bereits Lichter ein- und ausschalten und auch Temperatursensoren auslesen.

Datenbank

Um die Daten der Geräte (ID, angegebener Name, Status,...) zu speichern, haben wir uns eine Datenbank eingerichtet. Diese basiert auf MongoDB und ist ebenfalls ein wichtiger Bestandteil von Gaia. Alle Änderungen des Users werden festgehalten und in unserer Datenbank gespeichert. So muss bei der nächsten Nutzung nicht alles neu eingerichtet werden. Dank unserer Datenbank ist unser System persistent.

Dashboard

Unser Dashboard ist eine wichtige Seite unserer Plattform, da dort ausgewählte Widgets, und somit wichtige Informationen und Steuerungsfunktionen, ihren Platz finden. Wir haben in unserer Demo bereits einige Widgets auf unserem Dashboard platziert, um die Funktionalitäten besser illustrieren zu können.

Individuell anpassbares Dashboard

Unser Dashboard ist durch den Nutzer individuell anpassbar. Hier können je nach Belieben Widgets platziert werden. Wie viele Widgets von welcher Art angelegt werden sollen spielt keine Rolle. Der Kreativität sind keine Grenzen gesetzt. In unserem Beispiel (siehe Screenshot) haben wir von jedem unserer implementierten Widgets eins platziert. Von den Lichtwidgets haben wir sogar direkt drei Stück auf dem Dashboard platziert, um alle verfügbaren, steuerbaren Lichter bedienen zu können.

Widgets

Eine unserer Funktionalitäten stellen die Widgets dar. Sie sind kleine Elemente, welche individuell auf dem Dashboard platziert werden können. Mit ihnen kann man die Endgeräte ansteuern und somit beispielsweise häufig genutzte Lichter ganz bequem und schnell sichtbar auf dem Dashboard anzeigen lassen. Der Anordnung der Widgets und der Häufigkeit und Art sind keine Grenzen gesetzt. Momentan bietet Gaia folgende drei Widgets an:

Temperatur

Unser Temperaturwidget zeigt die aktuelle, lokale Temperatur an. Dabei werden sowohl der festgelegte Name, als auch die aktuelle Temperatur und in welcher Einheit gemessen wurde, dargestellt. In unserem Beispiel befindet sich der Temperatursensor in einem Wohnzimmer, er kann jedoch auch beispielsweise zur Messung der Außentemperatur genutzt werden.

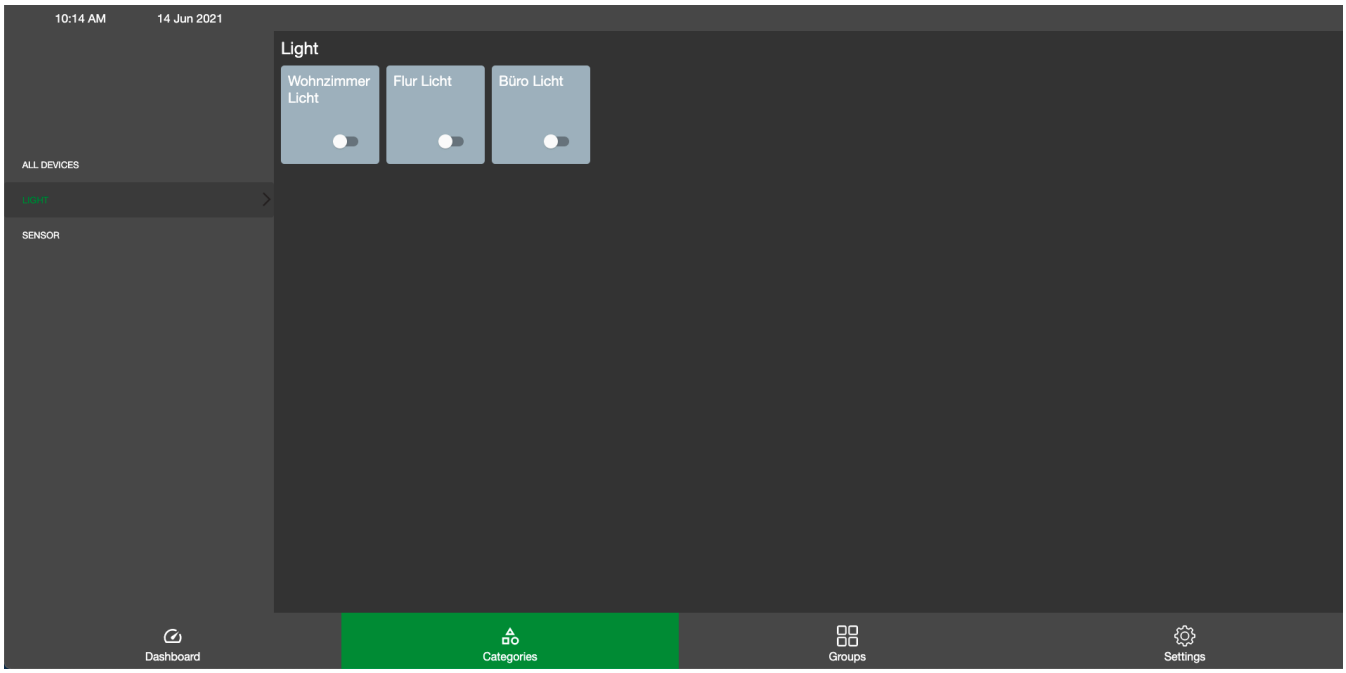
Wetter

Bei unserem Wetterwidget kann man sich das Wetter einer spezifischen Stadt anzeigen lassen. Es gibt sowohl die gewählte Stadt, als auch die Temperatur, die gefühlte Temperatur, die maximale und minimale Temperatur und die Luftfeuchtigkeit an. Die aktuellen Wetterdaten beziehen wir mithilfe einer API von openweathermap.org.

Licht

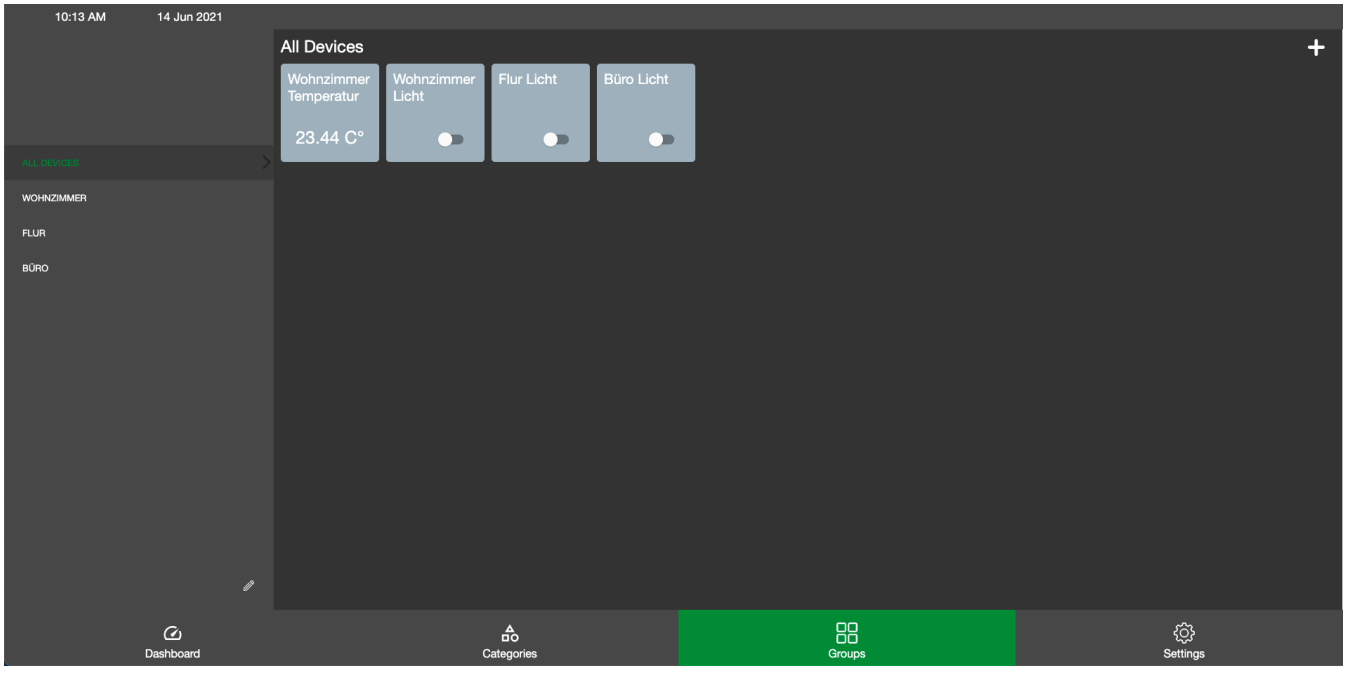
Mithilfe unseres Lichtwidgets kann man das angebundene Licht ein- und ausschalten. Es zeigt den Namen des hinterlegten Lichts und den Status, ob das Licht im Moment ein- oder ausgeschalten ist, an.

Kategorien



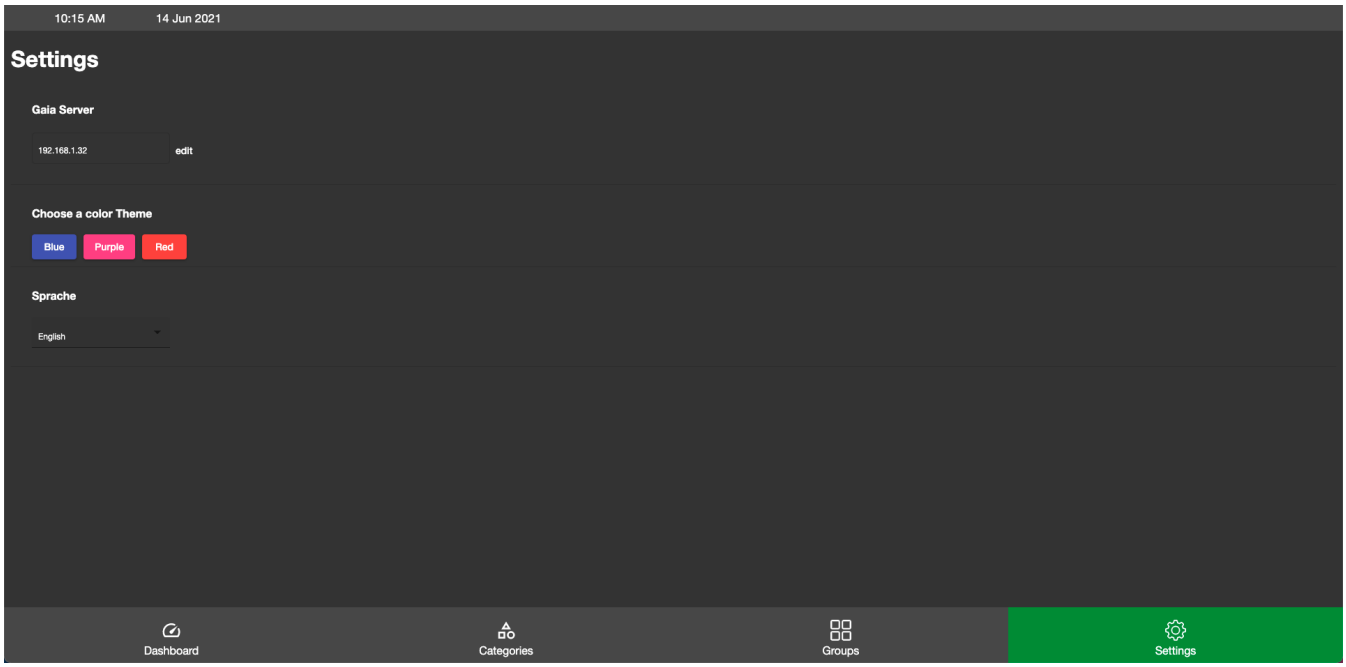
Auf unserer Kategorien-Seite kann man die automatisch erkannten und eingepflegten Geräte nach Kategorien sortiert anzeigen lassen. So werden die Geräte in Lichter und Sensoren unterteilt und geordnet.

Gruppen



Auf unserer Gruppen-Seite kann man sich selbst Gruppierungen der Geräte anlegen. Um in den Editiermodus zu wechseln, muss man in der linken Menüspalte auf das Stiftsymbol klicken. Nun kann man bereits angelegte Gruppen bearbeiten oder löschen und sich aber auch eigene Gruppen neu anlegen. Mithilfe von Gruppen fällt es den Usern leichter, ihre Geräte wiederzufinden. Man kann beispielsweise seine Ausstattung nach Zimmern (Wohnzimmer, Schlafzimmer, Küche,...) oder aber auch nach Aktivitäten (Filmeabend, Hausparty, Entspannung,...) sortieren und eine Gruppe erstellen. Somit fällt die lästige Suche nach bestimmten Geräten weg, und man hat alles in der gewünschten Ordnung.

Einstellungen



Bei unserer Einstellungs-Seite kann man verschiedene Einstellungen an der Applikation vornehmen.

Serveradresse

Das Feld der Serveradresse gibt an, auf welchem Server Gaia momentan läuft. Wenn es eine Erneuerung der Adresse geben sollte, kann man diese auch modifizieren. Um ein versehentliches Ändern der Serveradresse auszuschließen haben wir jedoch einen kleinen Sicherheitsmechanismus eingebaut. So muss der User bewusst auf den edit-Button klicken, um Änderungen an der Adresse vorzunehmen. Nachdem die Adresse abgeändert wurde, muss erneut auf den Button geklickt werden, um die Änderung zu bestätigen.

Themes

In unserer Designfindungsphase hatten wir uns die Möglichkeit gewünscht, das Farbschema der Applikation nach eigenen Bedürfnissen anpassen zu können. Leider konnte diese Funktion von uns nicht mehr umgesetzt werden, da eine umfassende Einarbeitung in die Thematik erforderlich war und wir diesen Anforderungen zeitlich nicht mehr gerecht werden konnten.

Sprache

Um Gaia zu Internationalisieren haben wir ebenfalls die Funktion der Sprachauswahl eingebaut. So kann die präferierte Sprache ausgewählt werden, damit nahezu jeder Gaia nutzen kann.

Was wir uns noch gewünscht hätten

Leider konnten wir nicht alle gewünschten Funktionalitäten auch umsetzen. Daher beschreiben wir nun, was wir uns zwar noch gewünscht hätten, aber leider nicht (mehr) realisieren konnten.

Alexa Integration

Die Alexa Integration war eines unserer Nice-to-have-Features, die es leider nicht mehr in die erste Version von Gaia geschafft haben. Wir hätten es schön gefunden, wenn man die Geräte auch über Sprachbefehle via Alexa hätte steuern können. Die Implementation dieser Funktion hätte allerdings deutlich mehr Zeit in Anspruch genommen. Da die Priorität dieser Funktion nicht so hoch war entschieden wir uns dazu, die Alexa Anbindung eventuell in einer späteren Version von Gaia zu implementieren.

Benutzerübersicht

Ein weiteres Feature, welches es leider nicht mehr geschafft hat war die Benutzerübersicht. Wir wollten die Möglichkeit anbieten, dass mehrere Benutzer auf Gaia agieren können. Dies wäre besonders sinnvoll für Familien, WGs oder andere Haushalte mit mehreren Bewohnern gewesen. Durch die Benutzerübersicht hätte jeder User seine eigenen Gruppen anlegen, und nach eigenen Wünschen gestalten, können.

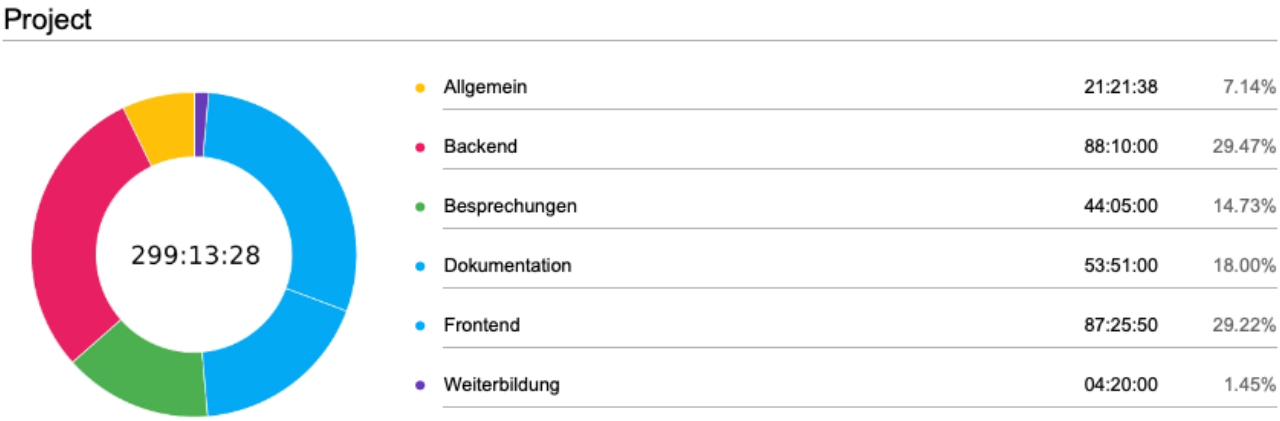
Webservice

Eine weitere Funktion welche wir uns für Gaia gewünscht hätten, und uns auch in Zukunft noch vorstellen können zu implementieren,

ist ein angebotener Webservice. Momentan ist es leider nur möglich, Gaia im lokalen Netzwerk zu nutzen. Daher hätten wir es gerne angeboten, Gaia von überall unterwegs steuern zu können.

Zeitjournal

Wir haben unsere Zeiten mit Clockify getrackt, da uns die vielfältigen Optionen und die benutzerfreundliche Oberfläche sehr gut gefallen haben. Außerdem stellte der gemeinsame Zugriff kein Problem dar wie es z.B. bei Excel oft der Fall ist. Clockify bot auch eine Dashboard-Funktion an bei welcher man schnell und einfach sehen konnte, wie lange wer an welchem Projekt gearbeitet hat. Hier eine Übersicht über unser Gesamtes Projekt. Es sind die einzelnen Buchungsposten ersichtlich. Die meiste Zeit hat das Backend in Anspruch genommen. Fast genauso lang haben wir am Frontend gearbeitet.



Zeitjournal der einzelnen Teammitglieder

Leia Lederer: Journal

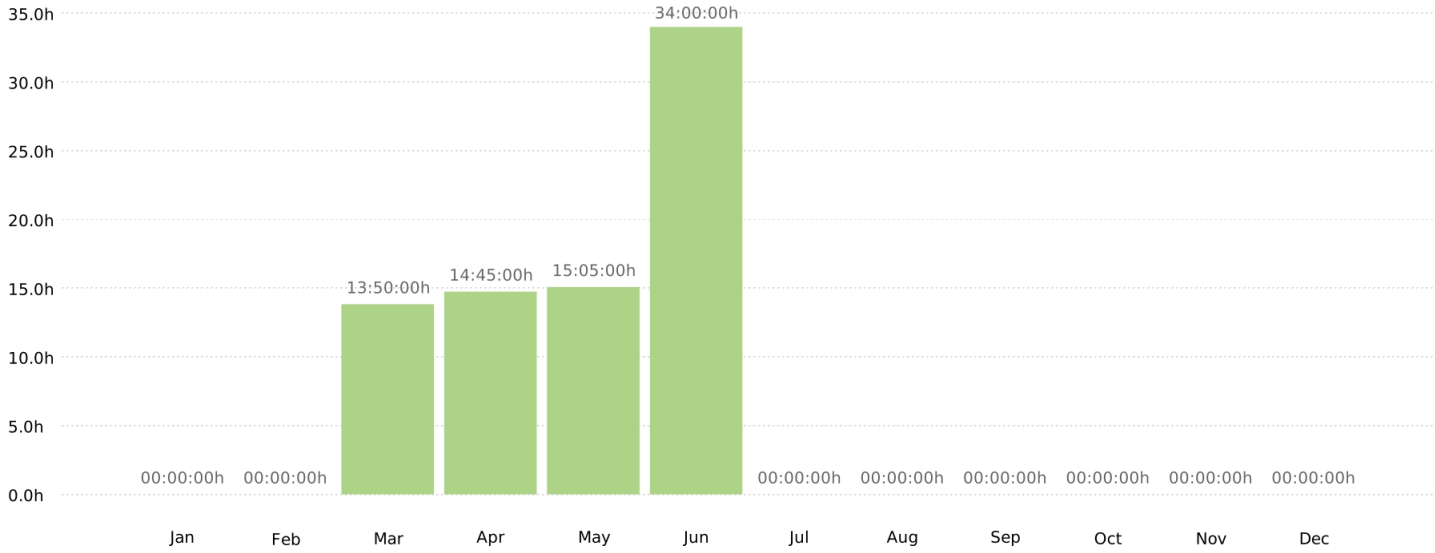
Marcel Henning: Journal

Marvin Klein Journal

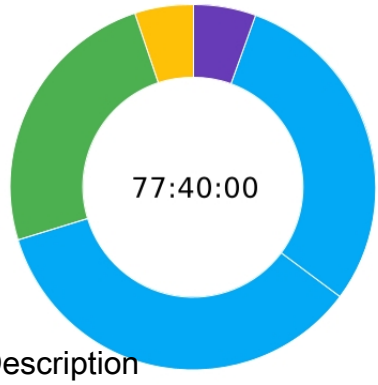
Summary report

01/01/2021 - 31/12/2021

Total: 77:40:00 Billable: 00:00:00 Amount: 0.00 USD

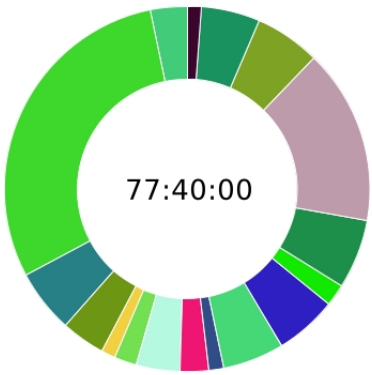


Project



Allgemein	04:00:00	5.15%
Besprechungen	19:05:00	24.57%
Dokumentation	27:15:00	35.09%
Frontend	23:00:00	29.61%
Weiterbildung	04:20:00	5.58%

Description



Plus-Button gestalten und positionieren	02:30:00	3.22%
Projektdoku	23:00:00	29.61%
Einarbeitung Angular	04:20:00	5.58%
Präsentation Generalprobe	03:00:00	3.86%
Child Group Menu Bindings	01:00:00	1.29%
Plus-Button und Icons	01:30:00	1.93%

●	Settingspage grober Entwurf	03:00:00	3.86%
●	Roadmap und Projekt einrichten	02:00:00	2.58%
●	Besprechung Projektdoku	01:00:00	1.29%
●	Lightwidget	04:15:00	5.47%
●	Projektdoku Feinschliff	04:15:00	5.47%
●	Child Group Menu	01:30:00	1.93%
●	Settingspage	04:45:00	6.12%
●	Meeting Vorlesung	12:05:00	15.55%
●	WeatherWidget	04:30:00	5.79%
●	Präsentation	04:00:00	5.15%
●	Projektidee Abstimmung	01:00:00	1.29%

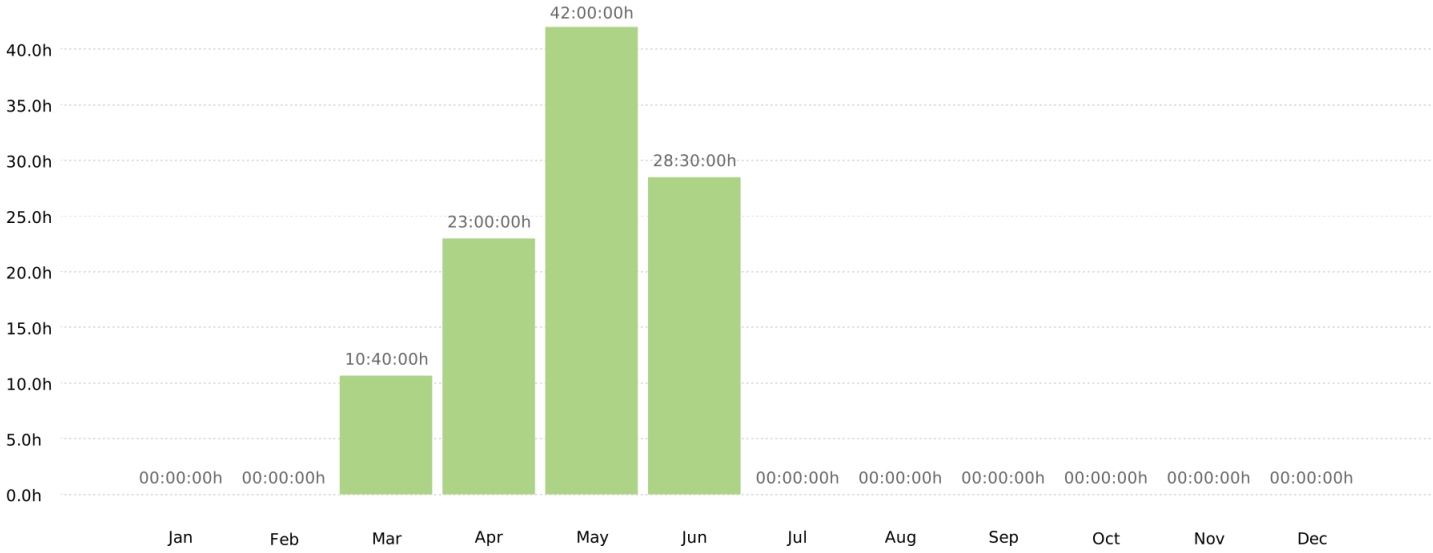
Project / Description	Duration	Amount
Allgemein	**04:00:00**	**0.00 USD**
Präsentation	04:00:00	0.00 USD
Besprechungen	**19:05:00**	**0.00 USD**
Präsentation Generalprobe	03:00:00	0.00 USD
Roadmap und Projekt einrichten	02:00:00	0.00 USD
Besprechung Projektdoku	01:00:00	0.00 USD
Meeting Vorlesung	12:05:00	0.00 USD
Projektidee Abstimmung	01:00:00	0.00 USD
Dokumentation	**27:15:00**	**0.00 USD**
Projektdoku	23:00:00	0.00 USD
Projektdoku Feinschliff	04:15:00	0.00 USD
Frontend	**23:00:00**	**0.00 USD**
Plus-Button gestalten und positionieren	02:30:00	0.00 USD

Child Group Menu Bindings	01:00:00	0.00 USD
Plus-Button und Icons	01:30:00	0.00 USD
Settingspage grober Entwurf	03:00:00	0.00 USD
Lightwidget	04:15:00	0.00 USD
Child Group Menu	01:30:00	0.00 USD
Settingspage	04:45:00	0.00 USD
WeatherWidget	04:30:00	0.00 USD
Weiterbildung	04:20:00	0.00 USD
Einarbeitung Angular	04:20:00	0.00 USD

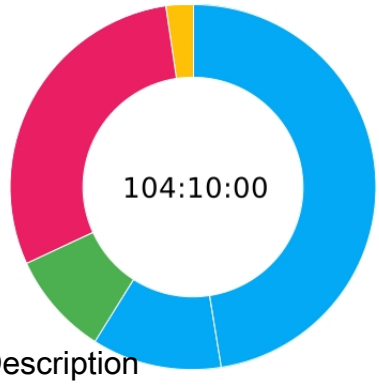
Summary report

01/01/2021 - 31/12/2021

Total: 104:10:00 Billable: 00:00:00 Amount: 0.00 USD

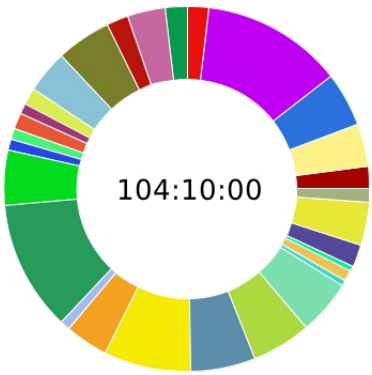


Project



Allgemein	02:30:00	2.40%
Backend	30:40:00	29.44%
Besprechungen	09:30:00	9.12%
Dokumentation	12:00:00	11.52%
Frontend	49:30:00	47.52%

Description



MQTT Client	02:00:00	1.92%
MQTT & GrahpQL Schnittstelle	03:30:00	3.36%
Vorbereitung Präsentation	02:00:00	1.92%
Meeting - Team Syncro	05:00:00	4.80%
UI changes	04:00:00	3.84%
Projektstruktur Planung	01:30:00	1.44%

Bug Fixing	01:00:00	0.96%
ESLint einrichten	01:30:00	1.44%
Projektidee Besprechung	01:00:00	0.96%
Projektbesprechung	01:00:00	0.96%
Codegen Client	05:00:00	4.80%
Dokumentation	12:00:00	11.52%
small server changes	01:00:00	0.96%
GraphQL Modules	03:55:00	3.76%
MQTT	08:00:00	7.68%
MobileUI	06:00:00	5.76%
UI	05:30:00	5.28%
Add Groups UI & fetch	05:00:00	4.80%
Team Syncro	00:30:00	0.48%
MQTT Spike	01:00:00	0.96%
Präsentation	00:30:00	0.48%
GraphQL erstes Schema	02:00:00	1.92%
Refactor LightWidget	04:00:00	3.84%
MQTT Lib	01:15:00	1.20%
Cleanup und ESLint hinzufügen	02:00:00	1.92%
Ui changes	04:00:00	3.84%
MQTT anfrage weiterleiten an graphql	05:00:00	4.80%
Daten Fetch vom Server	13:00:00	12.48%
Roadmap und Projekt einrichten	02:00:00	1.92%

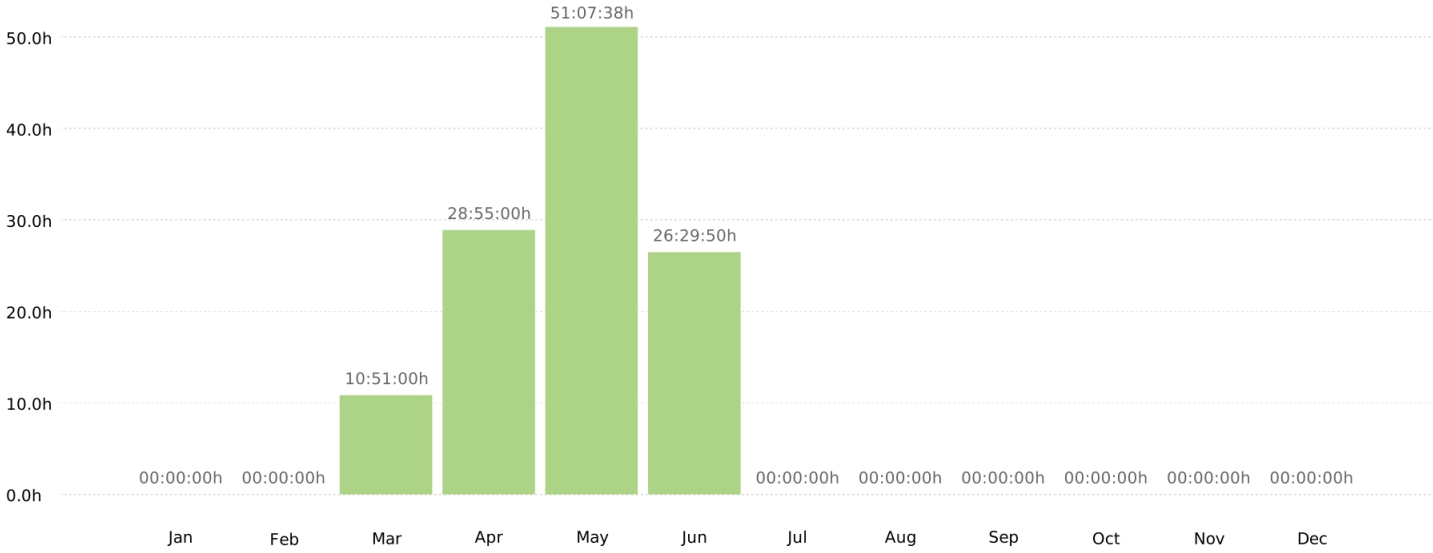
Project / Description	Duration	Amount
Allgemein	02:30:00	0.00 USD
Vorbereitung Präsentation	02:00:00	0.00 USD
Präsentation	00:30:00	0.00 USD
Backend	30:40:00	0.00 USD
MQTT Client	02:00:00	0.00 USD
MQTT & GrahpQL Schnittstelle	03:30:00	0.00 USD
Projektstruktur Planung	01:30:00	0.00 USD
ESLint einrichten	01:30:00	0.00 USD
small server changes	01:00:00	0.00 USD
GraphQL Modules	03:55:00	0.00 USD
MQTT	08:00:00	0.00 USD
MQTT Spike	01:00:00	0.00 USD
GraphQL erstes Schema	02:00:00	0.00 USD
MQTT Lib	01:15:00	0.00 USD
MQTT anfrage weiterleiten an graphql	05:00:00	0.00 USD
Besprechungen	09:30:00	0.00 USD
Meeting - Team Syncro	05:00:00	0.00 USD
Projektidee Besprechung	01:00:00	0.00 USD
Projektbesprechung	01:00:00	0.00 USD
Team Syncro	00:30:00	0.00 USD
Roadmap und Projekt einrichten	02:00:00	0.00 USD
Dokumentation	12:00:00	0.00 USD
Dokumentation	12:00:00	0.00 USD

Frontend	49:30:00	0.00 USD
UI changes	04:00:00	0.00 USD
Bug Fixing	01:00:00	0.00 USD
Codegen Client	05:00:00	0.00 USD
MobileUI	06:00:00	0.00 USD
UI	05:30:00	0.00 USD
Add Groups UI & fetch	05:00:00	0.00 USD
Refactor LightWidget	04:00:00	0.00 USD
Cleanup und ESLint hinzufügen	02:00:00	0.00 USD
Ui changes	04:00:00	0.00 USD
Daten Fetch vom Server	13:00:00	0.00 USD

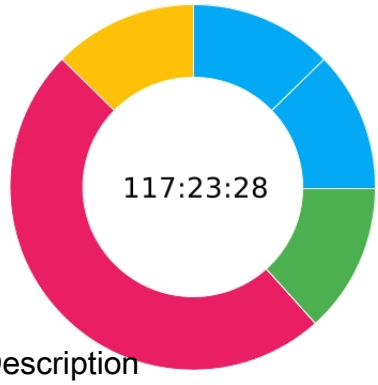
Summary report

01/01/2021 - 31/12/2021

Total: 117:23:28 Billable: 00:00:00 Amount: 0.00 USD

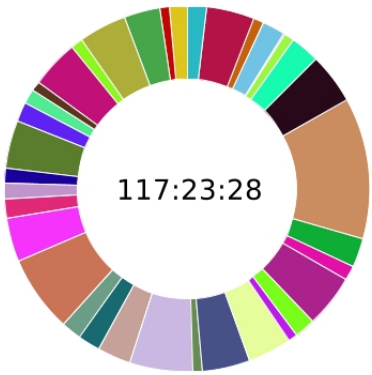


Project



Allgemein	14:51:38	12.66%
Backend	57:30:00	48.98%
Besprechungen	15:30:00	13.20%
Dokumentation	14:36:00	12.44%
Frontend	14:55:50	12.72%

Description



Groups	01:50:00	1.56%
Weekly	01:00:00	0.85%
Präsentation Generalprobe	03:39:00	3.11%
MQTT Kommunikation	05:00:00	4.26%
Button Bug gefixt	01:15:00	1.06%
#31 Groups	05:00:00	4.26%

Projekt Sync	01:00:00	0.85%
Planung der Struktur	01:30:00	1.28%
Besprechung Roadmap & Einrichtung	02:00:00	1.70%
#49 MQTT Firmware	05:00:00	4.26%
Branches aufgeräumt	01:30:00	1.28%
StartSkript RaspberryPi	01:40:00	1.42%
Präsentation	02:00:00	1.70%
#69 Codegen	04:30:00	3.83%
Weekly Sync	08:00:00	6.81%
Frontend Config	02:00:50	1.71%
Mqtt Publish Handler	02:25:00	2.06%
Implementierung MongoDB Connector	03:25:00	2.91%
#49 MQTT Handler	06:30:00	5.54%
Temperatursensor Handler	01:00:00	0.85%
Deployment	05:00:00	4.26%
#71 Codegen	04:30:00	3.83%
Besprechung Roadmap	01:00:00	0.85%
Struktur Apollo Server	02:10:00	1.85%
MQTTClient und GraphQL Schnitstelle	05:30:00	4.69%
Team Syncro	01:30:00	1.28%
MQTT Spike	03:00:00	2.56%
Dokumentation	14:36:00	12.44%
Frontend Categroy	05:20:00	4.54%
MQTT Broker	03:00:00	2.56%
Projektidee Abstimmung	01:00:00	0.85%
Clockify einrichten	00:11:00	0.15%
Temperatursensor	02:21:38	2.01%
#83 Handler Interface aktualisiert	01:00:00	0.85%

GraphIQL API Device & Resolver	05:00:00	4.26%
Server um Kategorie erweitert	02:00:00	1.70%

Project / Description	Duration	Amount
Allgemein	**14:51:38**	**0.00 USD**
Präsentation Generalprobe	03:39:00	0.00 USD
StartSkript RaspberryPi	01:40:00	0.00 USD
Präsentation	02:00:00	0.00 USD
Deployment	05:00:00	0.00 USD
Clockify einrichten	00:11:00	0.00 USD
Temperatursensor	02:21:38	0.00 USD
Backend	**57:30:00**	**0.00 USD**
MQTT Kommunikation	05:00:00	0.00 USD
#31 Groups	05:00:00	0.00 USD
Planung der Struktur	01:30:00	0.00 USD
#49 MQTT Firmware	05:00:00	0.00 USD
Branches aufgeräumt	01:30:00	0.00 USD
Mqtt Publish Handler	02:25:00	0.00 USD
Implementierung MongoDB Connector	03:25:00	0.00 USD
#49 MQTT Handler	06:30:00	0.00 USD
Temperatursensor Handler	01:00:00	0.00 USD
#71 Codegen	04:30:00	0.00 USD
Struktur Apollo Server	02:10:00	0.00 USD
MQTTClient und GraphQL Schnitstelle	05:30:00	0.00 USD

MQTT Spike	03:00:00	0.00 USD
MQTT Broker	03:00:00	0.00 USD
#83 Handler Interface aktualisiert	01:00:00	0.00 USD
GraphlQL API Device & Resolver	05:00:00	0.00 USD
Server um Kategorie erweitert	02:00:00	0.00 USD
Besprechungen	15:30:00	0.00 USD
Weekly	01:00:00	0.00 USD
Projekt Sync	01:00:00	0.00 USD
Besprechung Roadmap & Einrichtung	02:00:00	0.00 USD
Weekly Sync	08:00:00	0.00 USD
Besprechung Roadmap	01:00:00	0.00 USD
Team Syncro	01:30:00	0.00 USD
Projektidee Abstimmung	01:00:00	0.00 USD
Dokumentation	14:36:00	0.00 USD
Dokumentation	14:36:00	0.00 USD
Frontend	14:55:50	0.00 USD
Groups	01:50:00	0.00 USD
Button Bug gefixt	01:15:00	0.00 USD
#69 Codegen	04:30:00	0.00 USD
Frontend Config	02:00:50	0.00 USD
Frontend Categroy	05:20:00	0.00 USD