Smart-Garden – Dokumentation

Dokumentation Backendarchitektur

- Kommunikationsservice - GraphQL
- Message-oriented Middleware - RabbitMQ
- Microservicearchitektur

1. Einleitung und Motivation

In einer Zeit, in der Menschen immer weniger Zeit für traditionelle Gartenarbeit aufbringen können, wächst gleichzeitig das Interesse daran, die Herkunft unserer Lebensmittel besser nachzuvollziehen. Kurze Transportwege und regionale Produktion sind zentrale Faktoren nachhaltiger Ernährung. Doch ohne Automatisierung sind Hochbeete und Gartenhäuser häufig mit hohem Pflegeaufwand verbunden. Genau hier setzt Smart-Garden an: Es bietet eine IoT-basierte Lösung, um den Anbau von Gemüse und Kräutern sowohl im städtischen als auch im ländlichen Umfeld zu automatisieren und zugleich jederzeit transparent nachzuverfolgen.

2. Problem Statement

- **Zeitmangel:** Die meisten Freizeitgärtner haben nicht die Kapazitäten für regelmäßige Kontrolle und Pflege.
- **Nachhaltigkeit & Herkunft:** Lange Transportwege erhöhen CO_2-Emissionen; Konsumenten fordern Transparenz.
- **Know-how-Hürde:** Ohne automatisierte Unterstützung sind Fehlbewässerung und Schädlingsbefall häufige Probleme.

Problem Statement:

> „Wie kann eine modulare, skalierbare IoT-Plattform Hobbygärtnern und urbanen Gemeinschaften helfen, Hochbeete und Gartenhäuser eigenständig und energieeffizient zu betreiben, um Zeit zu sparen, Ressourcen zu schonen und das Wachsen der Lebensmittel digital nachzuverfolgen?“

3. Projektbeschreibung

Smart-Garden automatisiert Hochbeete und Gartenhäuser, indem es Sensordaten erfasst, diese in Echtzeit verarbeitet und auf Basis vordefinierter Regeln Aktionen an Aktoren auslöst. Die Nutzer sehen online alle Messwerte und Aktorenzustände, können manuell eingreifen und eigene Automatisierungsregeln erstellen. So lässt sich beispielsweise festlegen, dass bei einer Luftfeuchtigkeit unter 40 % und einer Bodenfeuchte unter 30 % automatisch die Bewässerungspumpe für eine festgelegte Zeit aktiviert wird.

4. Technische Architektur

4.1 IoT-Steuerung

Im Zentrum der gesamten Plattform steht die Sensor-Aktor-Kommunikation, die über standardisiertes MQTT-Protokoll realisiert ist. Auf Hardware-Seite kommen Mikrocontroller wie Arduino Uno zum Einsatz, an die jeweils eine beliebige Anzahl von Modulen angeschlossen werden kann. Diese Module unterteilen sich in:

- Sensoren (z. B. DHT11 für Temperatur und Luftfeuchtigkeit, kapazitive Bodenfeuchtesensoren, BH1750 für Beleuchtungsstärke)
- Aktoren (z. B. Relais-gesteuerte Wasserpumpen, Lüftungsmotoren, Status-LEDs)

4.1.1 Modul-Registrierung

1. **Initialisierung:** Beim Booten stellt jeder Mikrocontroller zuerst eine WLAN-Verbindung her.
2. **Registrierungsnachricht:** Anschließend versendet das Modul eine JSON-kodierte MQTT-Nachricht auf das zentrale Topic „register“:

```
{
  "moduleKey": "sg-f0f5bd525a98",
  "topics": {
    "temperature": "smart-garden/sg-f0f5bd525a98/temperature",
    "humidity": "smart-garden/sg-f0f5bd525a98/humidity",
    "moisture": "smart-garden/sg-f0f5bd525a98/moisture"
  }
}
```

3. **Backend-Empfang:** Der `ConnectionService` im Backend hört permanent auf dieses Topic. Erkennt er einen neuen `moduleKey`, so legt er einen Eintrag in der Datenbank an.

4.1.2 Laufende Datenerfassung

Sobald ein Modul registriert ist, geht es in den Regelbetrieb:

1. **Messzyklus:** Sensor-Module lesen zyklisch (z. B. alle 5 Sekunden) ihre Werte aus.
2. **JSON-Payload:** Jeder Datensatz wird als strukturierte JSON-Nachricht verpackt:

```
{
  "messageType": "State",
  "moduleKey": "sg-f0f5bd525a98",
  "moduleType": "Temperature",
  "min": 0,
  "max": 50,
  "unit": "°C",
  "stateType": "Continuous",
  "currentValue": 24.6
}
```

3. **MQTT-Publish:** Diese Nachricht wird auf dem zuvor zugewiesenen Topic veröffentlicht, etwa `smart-garden/sg-f0f5bd525a98/temperature`.
4. **Backend-Verarbeitung:** Der `ConnectionService` empfängt die Nachricht, validiert das Format und leitet sie intern an den `AutomationService` und die API-Services weiter. Dort werden die Daten in Echtzeit an das Frontend weitergereicht oder für die Überprüfung der Automatisierungsregeln verwendet.

4.1.3 Aktor-Steuerung

Aktoren bleiben im Leerlauf, bis sie über MQTT einen Steuerbefehl erhalten:

1. **Abonnements:** Jeder Aktor-Controller hat sich beim Start auf sein Action-Topic (z. B. `actuators/beet2-pump1/set`) subscribiert.

2. **Aktionsnachricht:** Ein Beispielbefehl könnte so aussehen:

```
{
  "moduleKey": "sg-f0f5bd525a98",
  "moduleType": "hatch",
  "actionKey": "hatch.open",
  "value": "50"
}
```

3. **Ausführung:** Durch den MQTT-Callback wird z.B. ein entsprechende Pin an dem der Aktor angeschlossen ist auf HIGH gesetzt.

4. **Monitoring:** Auch diese Statusupdates von Aktoren gehen zurück an den `ConnectionService` und werden von dort an Frontend und AutomationService verteilt.

```
{
  "messageType": "State",
  "actuatorKey": "sg-f0f5bd525a98",
  "actuatorType": "Hatch",
  "stateType": "Continuous",
  "min": 0,
  "max": 100,
  "unit": "%",
  "currentValue": 40
}
```

4.2 Hardware-Setup

1. **Mikrocontroller:**
   - Arduino Uno R4 WiFi
   - Es sind auch andere Boards möglich solange diese eine WiFi-Verbindung aufbauen können.

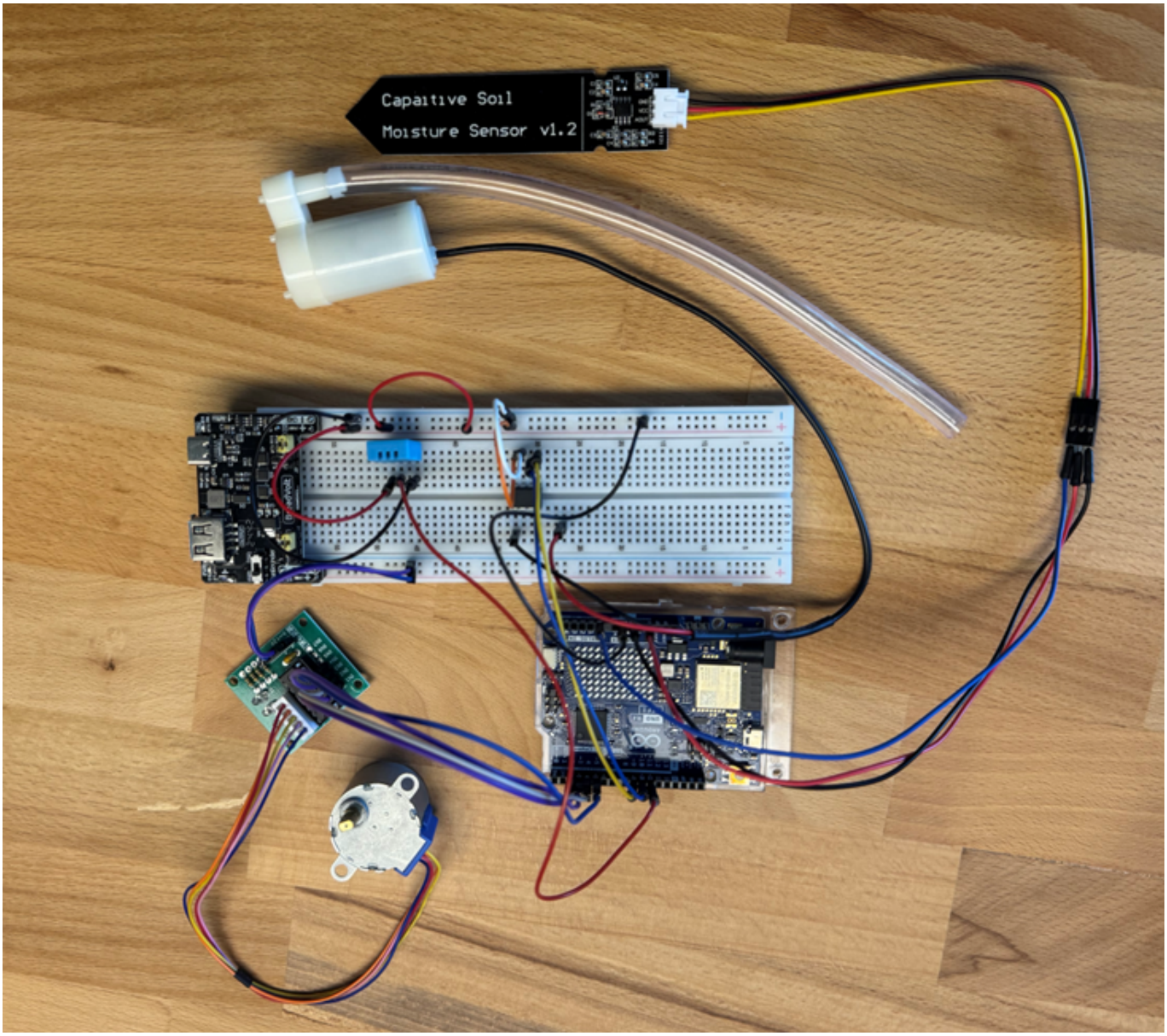
2. **Sensoren:**
   - Bodenfeuchtigkeitssensor (Capacitive soil moisture sensor)
   - DHT11 für Temperatur & Luftfeuchtigkeit

3. **Aktoren:**
   - Wasserpumpe
   - Schrittmotor für Belüftung

4. **Netzwerk:**
   - WLAN oder Ethernet-Shield

- MQTT-Broker (z. B. Mosquitto) lokal oder gehostet.

5. **Stromversorgung:**
    - Power Batterie Modul (5 V / 3,3 V)
    - USB-C Netzteil



4.3 Backend-Architektur

Das Backend von Smart-Garden und übernimmt drei zentrale Aufgaben: 1) Kommunikation mit den Modulen, 2) Umsetzung der Automatisierungsregeln und 3) die Anbindung und Echtzeit-Versorgung des Frontends. Durch klare Trennung in einzelne Services lassen sich Performance, Wartbarkeit und Skalierbarkeit optimieren.

4.3.1 ConnectionService

- **MQTT-Gateway:**: Baut beim Start eine dauerhafte Verbindung zu einem MQTT-Brokern (z. B. Mosquitto, EMQX) auf.

- **Modul-Lifecycle:**: Analysiert eingehende „register“-Topics, validiert JSON-Schemas und persistiert neue Module in der PostgreSQL-Tabelle.
- **Event-Bus:**: Für jede gültige Sensornachricht oder Aktorstatus meldet sich der ConnectionService über RabbitMQ

4.3.2 AutomationService

- **Regel-Repository:**
  - Lädt beim Start alle aktivierten Regeln aus einer Datenbanktabelle.
  - Jede Regel enthält:
    - **Bedingungsexpressionen** (z. B. „`temperature > 30 AND soilMoisture < 40`“),
    - **Aktionsdefinition** (z. B. „set actuators/beet2-pump1 ON for 30 s“),
- **Evaluierungs-Loop:**
  - Überprüft Intervallweise die Automatisierungsregeln
  - Bei erfüllter Regel erzeugt der AutomationService eine Aktionsnachricht über RabbitMQ.
  - Wertet Bedingungen per AST-Interpreter aus, um komplexe logische und arithmetische Ausdrücke zu unterstützen.

4.3.3 API-Layer & Frontend-Push

Das Backend stellt sowohl klassische REST-Endpoints als auch ein modernes GraphQL-API bereit und sorgt über Websocket-Mechanismen für Echtzeit-Updates im Frontend:

- **REST-API & GraphQL:**
  - **REST-Endpoints** für administrative Aufgaben und zum Abfragen der Module und Beete
  - **GraphQL-API** für flexible Abfragen und Mutationen aller Entitäten (Module, Beete, Regeln) sowie zur Strukturierung komplexer Abfragen in einem Request.
- **GraphQL Subscriptions & SignalR WebSockets:**
  - Das Frontend verbindet sich beim Laden wahlweise über SignalR-basierte WebSockets oder GraphQL Subscriptions.
  - Über diese persistente Verbindung werden Ereignisse in Echtzeit gepusht:
    - **Sensordaten:** Neue Messwerte pro Modul und Beet
    - **Aktorstatus:** Änderung von On/Off-Zustand, Laufzeitende

4.3.4 Nachrichtenfluss

Die folgenden Sequenzdiagramme veranschaulichen die Abläufe der Kommunikation im System:

Register

```
sequenceDiagram
    participant Module
```

```
    participant MQTT_Broker as MQTT Broker
    participant ConnectorService
    participant RMQ as RabbitMQ
    Module ->> MQTT_Broker: send register message
    MQTT_Broker ->> ConnectorService: register Device
    ConnectorService ->> RMQ: publish new Device
    RMQ ->> API: add new Device to list
```

Send Status Update

```
sequenceDiagram
    participant Module
    participant MQTT_Broker as MQTT Broker
    participant ConnectorService
    participant RMQ as RabbitMQ
    participant AutomationService
    participant API
    Module ->> MQTT_Broker: send state change
    MQTT_Broker ->> ConnectorService: update state
    ConnectorService ->> RMQ: publish state change
    RMQ ->> AutomationService: update state
    RMQ ->> API: update state
```

Ausführen von Actions

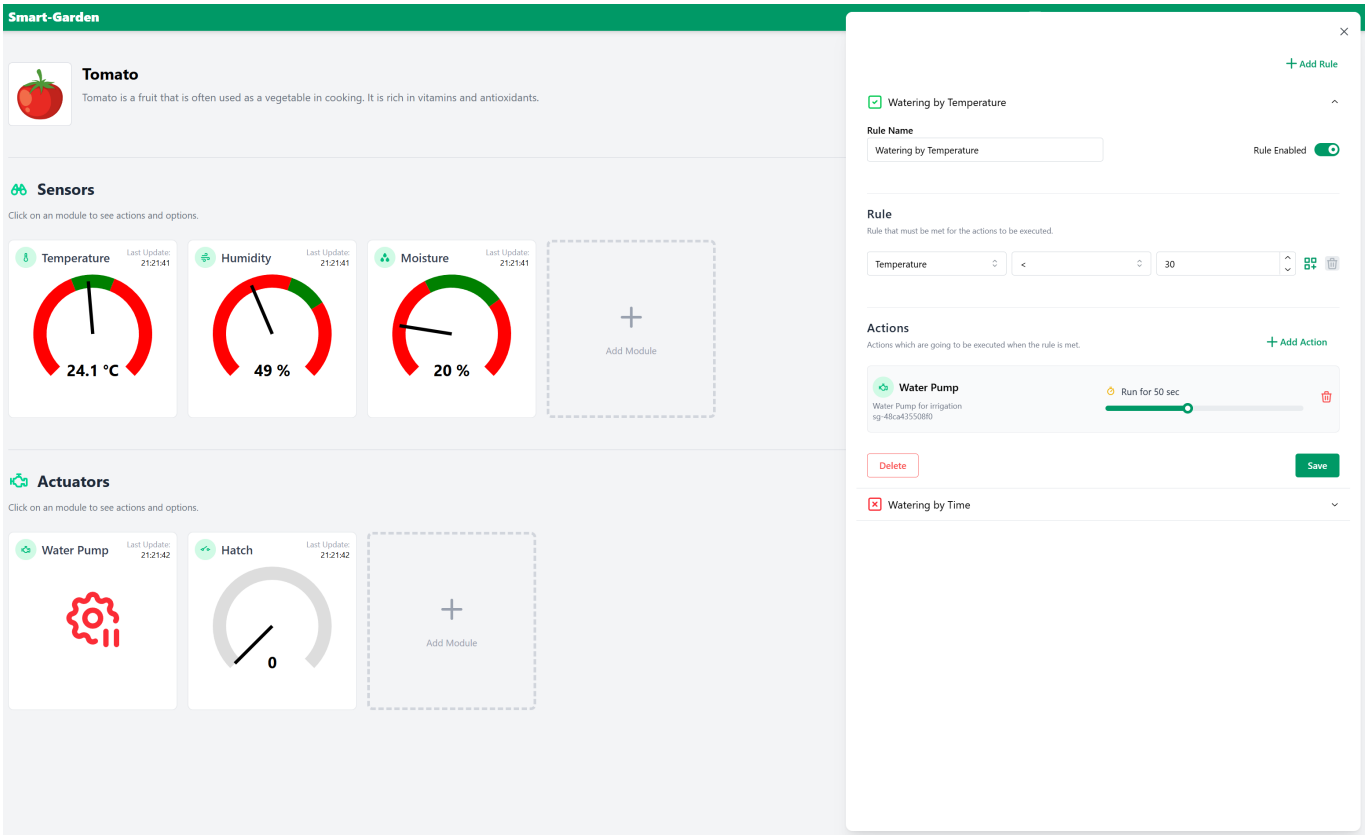
```
sequenceDiagram
    participant Module
    participant MQTT as MQTT Broker
    participant ConnectorService
    participant RMQ as RabbitMQ
    participant AutomationService
    participant API
    participant Frontend

    Frontend ->> API: Request Action-Execution
    API ->> RMQ: Publish Action-Execution
    AutomationService ->> RMQ: Publish Automation
Rule Action-Execution
    RMQ ->> ConnectorService: Receive Action-Execution
    activate ConnectorService
    ConnectorService ->> MQTT: Publish Action-Execution
to module1/waterpump
    deactivate ConnectorService
    MQTT ->> Module: Receive Execution

    activate Module
    Module -->> MQTT: Send Status Update
    deactivate Module
    MQTT -->> ConnectorService: Receive Status Update
```

```
    ConnectorService -->> RMQ: Publish State Change
    RMQ -->> AutomationService: Receive State Change
    RMQ -->> API: Receive State Change
    API -->> Frontend: Send SignalR / GraphQL Update
```

4.4 Frontend-Funktionalität

Das Webinterface erlaubt dem Nutzer:

- **Modulverwaltung:** Übersicht aller registrierten Sensor- und Aktormodule mit direkter Zuweisung zu einzelnen Beeten.
- **Live-Dashboard:** Anzeige aktueller Werte (z. B. 24 °C, 55 % Luftfeuchte, Bodenfeuchte 28 %) sowie Aktorstatus (Pumpe 1: AUS, Lüftung: 60 %).
- **Manuelle Steuerung:** Direkter Eingriff auf Aktoren wie Ein-/Aus-Schaltung der Pumpe oder Regelung der Motorstellung.
- **Regelgenerator:** Intuitive Oberfläche zum Erstellen, Bearbeiten und Aktivieren automatisierter Bewässerungs- und Belüftungsabläufe.



5. Implementierungsdetails

Initialisierung von Modulen

```
// References
extern WiFiClient network;
extern PubSubClient mqttClient;
extern String macAddress;
```

```
static String deviceId;
static SensorManager sensorManager;
static ActuatorManager actuatorManager;


void setup() {
  Serial.begin(9600);
  while (!Serial) {
    ;  // wait for serial port to connect. Needed for native USB port only
  }

  Serial.println("Connecting to WIFI ...");
  connectWifi();
  deviceId = getDeviceId();

  setupSensors();
  setupActuators();

  Serial.println("Connecting to MQTT ...");
  connectMQTT(deviceId);

  // Register sensors and actors at MQTT
  Serial.println("Registering Sensors ...");
  sensorManager.registerSensors(deviceId);
  Serial.println("Registering Actuators ...");
  actuatorManager.registerActuators(deviceId);


  mqttClient.setCallback(listen);
  actuatorManager.subscribeAll(mqttClient);
}

void loop() {
  mqttClient.loop();
  sensorManager.updateAll();
  actuatorManager.updateAll();
}
```

MQTT Initialisierung

```
void connectMQTT(String deviceId) {
  if (mqttClient.connected()) {
    mqttClient.disconnect();
  }

  mqttClient.setServer(mqttServerAddress, mqttServerPort);
  while (!mqttClient.connected()) {
    if (mqttClient.connect(deviceId.c_str())) {
      Serial.println("Success, MQTT device with ID " + deviceId + "
connected!");
    } else {
```

```
      Serial.print("failed, rc=");
      Serial.print(mqttClient.state());
      Serial.println(" try again in 1 seconds");
      delay(1000);
    }
  }
}

void delayWithLoop(int ms) {
  int parts = ms / 1000;

  for (int i = 0; i < parts; i++) {
    delay(1000);
    mqttClient.loop();
  }
}

void sendToMQTT(const String topic, const String json) {
  if (mqttClient.connected()) {
    mqttClient.loop();
    mqttClient.publish(topic.c_str(), json.c_str(), false);
  }
}

void sendToMQTTRetained(const String topic, const String json) {
  if (mqttClient.connected()) {
    mqttClient.loop();
    mqttClient.publish(topic.c_str(), json.c_str(), true);
  }
}
```

Sensor / Aktor Initialisierung

```
class ActuatorManager {
private:
  Actuator* actuators[MAX_ACTUATORS];
  int actuatorCount = 0;

public:
  void addActuator(Actuator* actuator) {
    if (actuator && actuatorCount < MAX_ACTUATORS) {
      actuators[actuatorCount++] = actuator;
    } else {
      Serial.println("ActuatorManager full or invalid actuator!");
    }
  }

  void initializeActuators() {
    for (int i = 0; i < actuatorCount; i++) {
      actuators[i]->initialize();
    }
```

```
  }

  void registerActuators(const String deviceId) {
    StaticJsonDocument<1024> doc;
    doc["moduleKey"] = deviceId;
    JsonObject topics = doc.createNestedObject("topics");
    for (int i = 0; i < actuatorCount; i++) {
      actuators[i]->appendTopicsTo(topics);
    }

    char buffer[1024];
    size_t len = serializeJson(doc, buffer, sizeof(buffer));
    if (len == 0) {
      Serial.println("registerActuators: Serialization failed or buffer too
small.");
    }
    Serial.println(buffer);

    sendToMQTT(ACTUATOR_REGISTER_TOPIC, buffer);
    Serial.println("Actuator Registered");
  }

  void updateAll() {
    for (int i = 0; i < actuatorCount; ++i) {
      if (actuators[i]) {
        actuators[i]->updateFast();
        if (actuators[i]->shouldUpdate()) {
          actuators[i]->update();
          actuators[i]->markUpdated();
        }
      }
    }
  }

  void onActionMessage(JsonDocument& doc) {
    // Check if the message is a command
    if (doc["messageType"].as<String>() == ACTION_MESSAGE_TYPE) {
      for (int i = 0; i < actuatorCount; ++i) {
        actuators[i]->onActionMessage(doc);
      }
    }
  }

  void subscribeAll(PubSubClient& client) {
    StaticJsonDocument<512> doc;
    JsonObject topics = doc.to<JsonObject>();

    for (int i = 0; i < actuatorCount; ++i) {
      actuators[i]->appendTopicsTo(topics);
    }

    for (JsonPair kv : topics) {
      const char* topic = kv.value();
      client.subscribe(topic);
```

```
      Serial.print("Subscribed to topic: ");
      Serial.println(topic);
    }
  }
};
```

Aktor / Sensor anhand Beispiel Pump

```
class Pump : public Actuator {
private:
  const String id;
  const int pinBI;
  const int pinFI;
  const String pumpTopic;

  String pumpState = "Stopped";
  unsigned long pumpStopTime = 0;
  bool pumpRunningForDuration = false;

public:
  Pump(int pinBI, int pinFI, String deviceId)
    : pinBI(pinBI),
      pinFI(pinFI),
      id(deviceId),
      pumpTopic("smart-garden/" + deviceId + "/waterpump") {

    Serial.print("Pump Topic: ");
    Serial.println(pumpTopic);
  }

  unsigned long getInterval() const override {
    return 5000;
  }

  void initialize() override {
    pinMode(pinBI, OUTPUT);
    pinMode(pinFI, OUTPUT);
  }

  void update() override {
    sendPumpStatus();
  }

  void updateFast() override {
    unsigned long now = millis();
    if (pumpRunningForDuration && now >= pumpStopTime) {
      setPump(false);
      pumpRunningForDuration = false;
    }
  }
```

```
  void appendTopicsTo(JsonObject& parent) override {
    parent["pump"] = pumpTopic;
  }

  void onActionMessage(JsonDocument& doc) override {
    Serial.println("Pump Action called");
    String moduleType = doc["moduleType"] | "";
    if (moduleType != "Pump") return;

    String actionKey = doc["actionKey"] | "";
    String actionType = doc["actionType"] | "";

    if (actionKey == "pump.start" && actionType == "Command") {
      setPump(true);
    } else if (actionKey == "pump.stop" && actionType == "Command") {
      setPump(false);
      pumpRunningForDuration = false;
    } else if (actionKey == "pump.run" && actionType == "Value") {
      float value = doc["value"].as<float>();
      if (value > 0.0f) {
        pumpStopTime = millis() + static_cast<unsigned long>(value * 1000);
        pumpRunningForDuration = true;
        setPump(true);
      }
    } else {
      Serial.println("Unknown action key or type");
    }
  }


private:

  void setPump(bool on) {
    if (on) {
      digitalWrite(pinBI, HIGH);
      digitalWrite(pinFI, LOW);
      pumpState = "Running";
    } else {
      digitalWrite(pinBI, LOW);
      digitalWrite(pinFI, LOW);
      pumpState = "Stopped";
    }

    sendPumpStatus();
  }

  void sendPumpStatus() {
    JsonDocument doc;
    doc["messageType"] = "State";
    doc["moduleKey"] = id;
    doc["moduleType"] = "Pump";
    doc["stateType"] = "Discrete";
    doc["state"] = pumpState;
```

```
      char buffer[512];
      serializeJson(doc, buffer);

      sendToMQTTRetained(pumpTopic, buffer);

      Serial.print("Pump Status: ");
      Serial.println(buffer);
    }
  };
```

6. Fazit

Smart-Garden zeigt, wie moderne IoT-Lösungen vor dem Hintergrund nachhaltiger Lebensmittelproduktion Mehrwert schaffen können. Mithilfe dieser Architektur ist es ein leichtes neue Aktoren und Sensoren zu erweitern.

MQTT ermöglicht sehr einfach das Senden und Empfangen von Nachrichten von und zu IoT-Devices und die Verarbeitung im Backend.