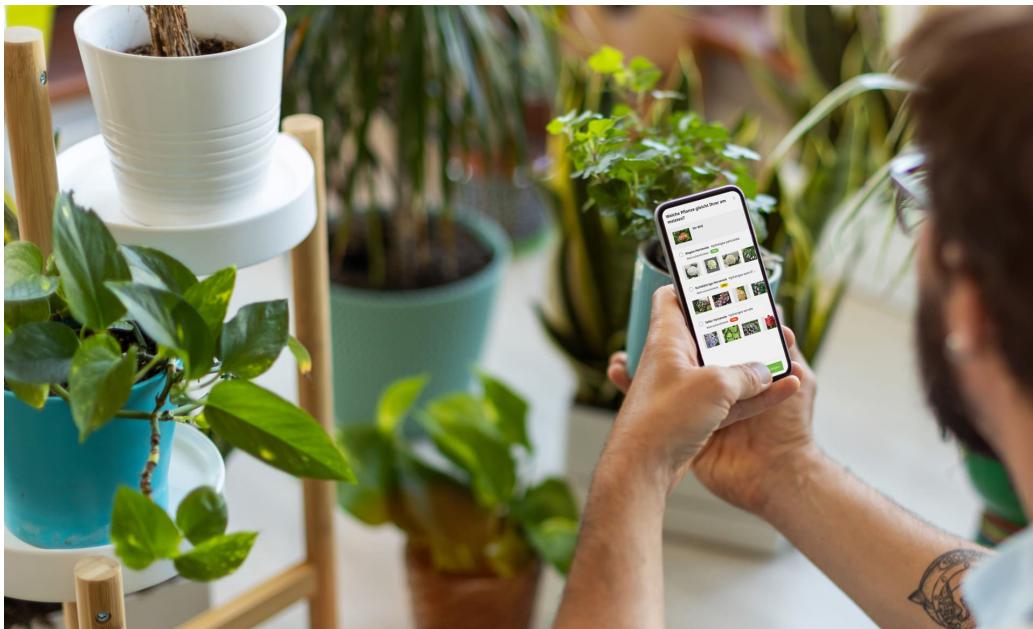


MoistureMate

Softwareentwurf und Anwendung verteilter Systeme



Projekt I

Dieses Projekt wurde über das ganze Semester weiter entwickelt und als Grundlage für unser Abschlussprojekt verwendet.

Der ESP8266-Mikrocontroller ermöglicht es, den Feuchtigkeitsgehalt in Echtzeit zu überwachen und den Zustand einer Pflanze zu erkennen. Durch den Zugriff auf die Sensorwerte über einen Webbrowser kann der Benutzer das Feuchtigkeitsniveau visuell überprüfen und feststellen, wann es erforderlich ist, die Pflanze zu gießen. Dadurch wird eine präzise Kontrolle und Pflege der Pflanze ermöglicht, um sicherzustellen, dass sie stets den optimalen Feuchtigkeitszustand aufweist.

1

DATA

Die Sensordaten werden an einen Webserver gesendet

2

WEBSERVER

Wird von dem Arduino gehostet und ist über ein lokales Netz erreichbar

3

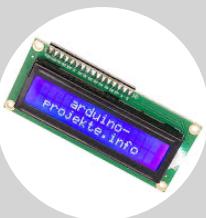
REST API

Über Rest APIs werden Sensor Daten abgerufen und vom Aktor gesteuert



ESP 8266

Die Verbindung zum Wi-Fi Netzwerk ermöglicht den Zugriff auf die Sensorwerte.



Aktor LCD Bildschirm

Die Feuchtigkeitswerte der Pflanze werden hier angezeigt.



Feuchtigkeitssensor

Die Feuchtigkeitswerte können mit diesem Sensor erfasst werden.

Erste Schritte

Der Code stellt eine Verbindung zu einem Wi-Fi-Netzwerk her und startet einen Webserver auf dem ESP8266. Der Feuchtigkeitssensor misst den Feuchtigkeitswert und zeigt ihn auf dem LCD-Display an. Gleichzeitig werden die Messwerte als JSON-Daten über den Webserver bereitgestellt.



```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <ArduinoJson.h>
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WebServer.h>
```

Diese Bibliotheken bieten die grundlegenden Funktionen und Protokolle, die für die Implementierung des Feuchtigkeitssensors und des Webservers benötigt werden. Sie vereinfachen die Entwicklung, indem sie vorgefertigte Funktionen und Methoden bereitstellen, um die Konfiguration des ESP8266, die Kommunikation mit dem Sensor und die Interaktion mit dem Webbrowser zu erleichtern.



Wire: Diese Bibliothek ermöglicht die Kommunikation über den I2C-Bus und wird verwendet, um den LCD-Bildschirm anzusteuern.

LiquidCrystal_I2C: Diese Bibliothek erleichtert die Ansteuerung von LCD-Displays über den I2C-Bus.

ArduinoJson: Diese Bibliothek bietet Funktionen zum Lesen und Schreiben von JSON-Daten und wird verwendet, um Sensorwerte im JSON-Format zu verarbeiten.

ESP8266WiFi: Diese Bibliothek ermöglicht die Verbindung und Kommunikation des ESP8266 mit einem WLAN-Netzwerk.

WiFiClient: Diese Bibliothek stellt Funktionen für die Verbindungsherstellung mit einem Server über das WLAN bereit.

SP8266WebServer: Diese Bibliothek ermöglicht die Erstellung eines Webservers auf dem ESP8266-Mikrocontroller, um Webseiten und HTTP-Anfragen zu verarbeiten.

Verbindung zum Wi-Fi-Netzwerk

In ersten Code-Abschnitt werden die Wi-Fi-Verbindungsinformationen konfiguriert und die Verbindung zum Wi-Fi-Netzwerk hergestellt

```
const char* ssid = "hfg-gast";
const char* password = "3FarbenGrau";
const int sensorPin = A0;
```

SSID und Passwort: Die Wi-Fi-Verbindungsinformationen sowie der Pin für den Sensor definiert.

```
LiquidCrystal_I2C lcd(0x27, 20, 4);
ESP8266WebServer server(80);
String jsonData;
```

Webserver Port: Ein Webserver auf Port 80 erstellt und eine leere Zeichenkette für die JSON-Daten deklariert.

```
void initWiFi() {
    WiFi.mode(WIFI_STA);
    WiFi.begin(ssid, password);

    Serial.println("Connecting to WiFi ..");
    while (WiFi.status() != WL_CONNECTED) {
        Serial.print(".");
        delay(100);
    }
}
```

Verbindung: Die Funktion **initWiFi()** initialisiert die Wi-Fi-Verbindung des ESP8266 im Station-Modus (**WIFI_STA**). Sie versucht, sich mit dem angegebenen Wi-Fi-Netzwerk zu verbinden. Während des Verbindungsaufbaus wird der Status überprüft, und es wird eine Wartezeit von 100 Millisekunden zwischen den Überprüfungen eingefügt. Die Funktion blockiert den Codefluss, bis die Verbindung erfolgreich hergestellt ist (**Status "WL_CONNECTED"**).

Nachrichten im JSON-Format

Das Abrufen der aktuellen Sensorwerte wird ermöglicht und das Senden als JSON-formatierte HTTP-Antwort. Ein Empfangen von Nachrichten über eine HTTP-POST-Anfrage sollte ermöglicht werden, die Gültigkeit des empfangenen JSON-Formats wird überprüft. Es sollte sichergestellt werden, dass Nachrichten extrahiert und der LCD-Bildschirm aktualisiert wird.

```

void getSensorData() {
    server.send(200, "text/json", jsonData);
}

void postMessage() {
    if (!server.hasArg("plain")) {
        server.send(400, "text/plain", "Bad Request: Missing request body");
        return;
    }

    String jsonMessage = server.arg("plain");

    StaticJsonDocument<128> jsonDoc;
    DeserializationError error = deserializeJson(jsonDoc, jsonMessage);
}

```

JSON Data: Die Funktion **getSensordata()** ermöglicht das Abrufen der aktuellen Sensorwerte und sendet sie als JSON-formatierte HTTP-Antwort. Die Funktion **postMessage()** ermöglicht das Empfangen von Nachrichten über eine HTTP-POST-Anfrage, überprüft die Gültigkeit des empfangenen JSON-Formats, extrahiert die Nachricht und aktualisiert den LCD-Bildschirm entsprechend. Die Zeile **String jsonMessage = server.arg(„plain“)** liest den Inhalt des HTTP-Anfrage-Bodys mit dem Namen "plain" aus, der erwartungsgemäß eine JSON-Zeichenkette enthält.

```

if (error) {
    server.send(400, "text/plain", "Bad Request: Invalid JSON format");
    return;
}

if (!jsonDoc.containsKey("message")) {
    server.send(422, "text/plain", "Unprocessable Entity: Missing 'message' key in JSON");
    return;
}

String message = jsonDoc["message"].as<String>();

if (message.length() == 0) {
    server.send(422, "text/plain", "Unprocessable Entity: 'message' value cannot be empty");
    return;
}

```

Überprüfung: Mit **if (error)** wird überprüft ob Fehler aufgetreten sind bei der Deserialisierung der JSON-Daten, wenn das der Fall ist wird ein 400er gesendet. Die Funktion **String message = jsonDoc[„message“].as<String>()** extrahiert den Inhalt des Feldes "message" aus den JSON-Daten und speichert ihn in der Variablen message. **If (message.length() == 0)** überprüft ob die Länge der Nachricht gleich Null ist und es wird ein Statuscode 422 ausgegeben wenn die Nachricht leer ist.

Anzeigen der Nachrichten

Mit Hilfe einer HTML-Seite können die Sensordaten nun auch im Browser abgerufen werden und nicht nur im seriellen Monitor der Arduino Entwicklungsumgebung.

```
void serveSensorPage() {
    String html = R"(
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Sensor Data</title>
    <script>
        function fetchSensorData() {
            fetch('/sensorData')
                .then(response => response.json())
                .then(data => {
                    document.getElementById('rawValue').innerHTML = data.rawValue;
                    document.getElementById('moisturePercentage').innerHTML = data.moisturePercentage;
                })
                .catch(error => {
                    console.error('Error fetching sensor data:', error);
                });
        }
        setInterval(fetchSensorData, 2000);
    </script>
</head>
<body>
    <h1>Sensor Data</h1>
    <p>Raw Value: <span id="rawValue"></span></p>
    <p>Moisture %: <span id="moisturePercentage"></span></p>
</body>
</html>
)";

    server.send(200, "text/html", html);
}
```

Webpage: Dabei wird mit JavaScript über eine HTTP-Anfrage **fetch()** die aktuellen Sensorwerte vom Server abgerufen und in den entsprechenden HTML-Elementen aktualisiert.

Konfiguration des REST-Servers und der Routen

Diese Funktion ermöglicht es dem Webserver, auf verschiedene Anfragen zu reagieren und entsprechende Aktionen auszuführen, wie das Anzeigen von HTML-Seiten, Abrufen von Sensorwerten oder Verarbeiten von Nachrichten.

```
void restServerRouting() {  
    server.on("/", HTTP_GET, []() {  
        server.send(200, "text/html", "Welcome to the REST Server");  
    });  
    server.on("/sensor", HTTP_GET, serveSensorPage);  
    server.on("/sensorData", HTTP_GET, getSensorData);  
    server.on("/message", HTTP_POST, postMessage);  
    server.onNotFound([]() {  
        server.send(404, "text/plain", "Resource not found");  
    });  
}
```

Routing: Mit Hilfe der Funktion `restServerRouting()` werden die Routen für den Webserver festgelegt.

Kontinuierliche Erfassung und Berechnung von Sensorwerten

```
void loop() {  
    int sensorValue = analogRead(sensorPin);  
    int moisturePercentage = map(sensorValue, 0, 1023, 100, 0);
```

Routing: Der aktuelle Sensorwert wird mit der Funktion `analogRead(sensorPin)` ausgelesen und in der Variable `sensorValue` gespeichert. Der Sensorwert wird mit der Funktion `map(sensorValue, 0, 1023, 100, 0)` auf einen Feuchtigkeitsprozentsatz umgerechnet und in der Variable `moisturePercentage` gespeichert.

```
lcd.setCursor(0, 1);
lcd.print("Raw Value: ");
lcd.print(sensorValue);
lcd.setCursor(0, 2);
lcd.print("Moisture %: ");
lcd.print(moisturePercentage);

server.handleClient();
delay(2000);
}
```

Anzeige: Die aktuellen Sensorwerte (roher Wert und Feuchtigkeitsprozentsatz) auf dem LCD-Bildschirm angezeigt. Anschließend wird die Funktion `server.handleClient()` aufgerufen, um eingehende Anfragen an den Server zu verarbeiten, und es folgt eine Verzögerung von 2 Sekunden, bevor die Schleife erneut durchlaufen wird.

Projekt II

Für die Endpräsentation wurde unser Projekt dann mit Azure verbunden und die Website ausgebaut. Es ist jetzt möglich die Sensordaten über Azure anzeigen zu lassen und über die Website Nachrichten an den LCD-Bildschirm zu senden.



Initialisierung des Azure IoT Hub-Clients

```
static void initializeClients()
{
    az_iot_hub_client_options options = az_iot_hub_client_options_default();
    options.user_agent = AZ_SPAN_FROM_STR(AZURE_SDK_CLIENT_USER_AGENT);

    wifi_client.setTrustAnchors(&cert);
    if (az_result_failed(az_iot_hub_client_init(
        &client,
        az_span_create((uint8_t*)host, strlen(host)),
        az_span_create((uint8_t*)device_id, strlen(device_id)),
        &options)))
    {
        Serial.println("Failed initializing Azure IoT Hub client");
        return;
    }
}
```

Initialisierung: Die Funktion `initializeClients()` initialisiert den Azure IoT Hub-Client. Sie verwendet die `az_iot_hub_client_init()` Funktion, um den Client mit den erforderlichen Informationen wie Host, Geräte-ID und Optionen zu konfigurieren.

Verbindung zu Azure IoT Hub über MQTT

```
if (mqtt_client.connect(mqtt_client_id, mqtt_username, sas_token))
{
    Serial.println("connected.");
}
```

Verbindung: Die Funktion `connectToAzureIoTHub()` stellt eine Verbindung zu Azure IoT Hub über MQTT her. Sie verwendet die `mqtt_client.connect()` Funktion, um sich mit Azure IoT Hub zu verbinden.

Verbindung zu Azure IoT Hub über MQTT

```
29 |     document.getElementById("sendButton").addEventListener("click", () => {
30 |       const message = document.getElementById("messageInput").value;
31 |
32 |       fetch("/sendMessage", {
33 |         method: "POST",
34 |         headers: {
35 |           "Content-Type": "application/json",
36 |         },
37 |         body: JSON.stringify({ message }),
38 |       }).catch(error) => console.error("Error:", error));
39 |     });

```

Send: Der Code reagiert auf einen Klick auf den "Senden"-Button, indem er den Inhalt des Eingabefelds mit der ID "messageInput" an den Server sendet. Die Daten werden als JSON formatiert und mithilfe der `fetch`-Funktion in einer POST-Anfrage an den Endpunkt "/sendMessage" gesendet. Bei auftretenden Fehlern wird eine entsprechende Fehlermeldung in der Konsole ausgegeben.

Interval-Funktion für periodische

```
41 |     setInterval(() => {
42 |       fetch("/db")
43 |         .then(response) => response.json()
44 |         .then(data) => [
45 |           const moisturePercentage = data.moisturePercentage;
46 |           document.getElementById("moisturePercentage").textContent =
47 |             "Moisture Percentage: " + moisturePercentage;
48 |           const lastUpdated = data.lastUpdated;
49 |           document.getElementById("lastUpdated").textContent =
50 |             "Last Updated: " + new Date(lastUpdated).toLocaleString();
51 |         ]
52 |         .catch(error) => console.error("Error:", error));
53 |     }, 10000);
54 |   </script>
55 | </body>
56 | </html>
57 | 
```

Diese Funktion ruft alle 10 Sekunden den Endpunkt "/db" auf dem Server auf, um Daten abzurufen. Die empfangenen JSON-Daten werden in JavaScript-Objekte umgewandelt. Danach werden zwei HTML-Elemente aktualisiert: Das Element mit der ID "moisturePercentage" wird mit dem Wert von **moisturePercentage** aktualisiert, und das Element mit der ID "lastUpdated" wird mit einem formatierten Datumswert aktualisiert. Fehler werden in der Konsole ausgegeben.

App.js

Modulimport für Azure IoT Hub-Kommunikation

```
3  const Client = require("azure-iothub").Client;
4  const Message = require("azure-iot-common").Message;
```

Um die Kommunikation mit dem Azure IoT Hub zu ermöglichen, werden in den folgenden Zeilen die erforderlichen Module importiert: **azure-iothub** und **azure-iot-common**

Modulimport für Azure IoT Hub-Kommunikation

```
35  var request = new sql.Request();
36  request.query(
37    "SELECT TOP 1 rawValue, moisturePercentage FROM [dbo].[new-table] ORDER BY EventProcessedUtcTime DESC",
38    function (err, recordset) {
39      if (err) console.log(err);
40      res.status(200).json(recordset.recordset[0]);
41    }
42  );
43 });
44 );
```

In dieser Codezeile wird eine SQL-Abfrage an die Datenbank gesendet, um den neuesten Datensatz aus der Tabelle `[dbo].[new-table]` abzurufen. Das Ergebnis wird als JSON-Objekt an den Client zurückgegeben, wobei nur der erste Datensatz verwendet wird.

Senden einer Nachricht an den Azure IoT Hub

```
60 |     var data = JSON.stringify({
61 |       state: req.body.message,
62 |       name: "sensor",
63 |     });
```

In diesen Codezeilen wird eine Nachricht an den Azure IoT Hub gesendet, um mit einem verbundenen Gerät zu kommunizieren. Zuerst wird ein JSON-Datenobjekt erstellt, das Informationen enthält, die an das Gerät gesendet werden sollen. Anschließend wird die Nachricht mit den Daten initialisiert und über den IoT Hub-Client an das Zielgerät gesendet. Dadurch

Client.js

Aktualisierung der Uhrzeit-Anzeige

```
3  document.addEventListener("DOMContentLoaded", function () {
4    setInterval(function () {
5      var currentTime = new Date();
6      var hours = currentTime.getHours().toString().padStart(2, "0");
7      var minutes = currentTime.getMinutes().toString().padStart(2, "0");
8      var seconds = currentTime.getSeconds().toString().padStart(2, "0");
9      document.getElementById("currentTime").innerText =
10        |   hours + ":" + minutes + ":" + seconds;
11    }, 1000);
12  });
13
```

In diesem Code wird die aktuelle Uhrzeit angezeigt, indem das **DOMContentLoaded-Ereignis** abgefangen wird. Dann wird ein Intervall-Timer erstellt, der die aktuelle Uhrzeit alle Sekunde aktualisiert und in das entsprechende HTML-Element (**currentTime**) einfügt.

Datenbankabfrage

```
14  function loadDataBase() {
15    fetch("/db", { method: "GET" })
16      .then((response) => response.json())
17      .then((data) => {
18        | document.getElementById("sensorValue").innerHTML = data.rawValue;
19      });
20  }
21
```

Die Funktion **loadDataBase()** ruft Daten aus der Datenbank ab, indem eine GET-Anfrage an den **/db-Endpunkt** gesendet wird. Die erhaltenen Daten werden dann in das entsprechende HTML-Element (**sensorValue**) eingefügt.

Klick-Event-Handler

```
22  const button = document.getElementById("myButton");
23  button.addEventListener("click", function (e) {
24    console.log("myButton button was clicked");
25    fetch("/db", { method: "GET" })
26      .then(function (response) {
27        | if (response.ok) return response.json();
28        | throw new Error("Request failed.");
29      })
30      .then(function (data) {
31        | document.getElementById("counter").innerHTML = ` Data: ${JSON.stringify(
32        |   data
33        | )}`;
34      })
35      .catch(function (error) {
36        | console.log(error);
37      });
38  });

```

Wenn der Button geklickt wird, wird eine GET-Anfrage an den **/db-Endpunkt** gesendet, und die erhaltenen Daten werden verarbeitet und in das HTML-Element **counter** eingefügt.

Klick-Event-Handler zum Nachrichten senden

```
40 const inputTextField = document.getElementById("messageInput");
41
42 const sendButton = document.getElementById("sendButton");
43 console.log(sendButton);
44 sendButton.addEventListener("click", function (e) {
45   console.log("sendButton button was clicked");
46
47   console.log("this is a test");
48   fetch("/sendMessage", {
49     method: "POST",
50     body: JSON.stringify({
51       text: inputTextField.innerText,
52     }),
53     headers: {
54       "Content-Type": "application/json",
55     },
56   })
57     .then(function (response) {
58       if (response.ok) {
59         console.log("click was recorded");
60         return;
61       }
62       throw new Error("Request failed.");
63     })
64     .catch(function (error) {
65       console.log(error);
66     });
67 });
68
```

Ein Klick-Event-Handler wird für den Button **sendButton** hinzugefügt. Wenn der Button geklickt wird, wird eine POST-Anfrage an den **/sendMessage-Endpunkt** gesendet. Der Inhalt des Eingabefelds (**inputTextField**) wird als JSON-Datenobjekt im Anfragekörper gesendet.

