Hochschule Offenburg
offenburg.university

Embedded Frontend-Entwicklung mit .Net Core und Blazor

William Mendat

BACHELORARBEIT

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

Studiengang Angewandte Informatik

Fakultät Elektrotechnik, Medizintechnik und Informatik
Hochschule für Technik, Wirtschaft und Medien Offenburg

28.02.2022

Durchgeführt bei der Firma Junker Technologies

Betreuer
Prof. Dr.-Ing. Daniel Fischer, Hochschule Offenburg
M.Sc. Adrian Junker, Junker Technologies

Mendat, William:
Embedded Frontend-Entwicklung mit .Net Core und Blazor / William Mendat. – BACHELORARBEIT, Offenburg: Hochschule für Technik, Wirtschaft und Medien Offenburg, 2022. 44 Seiten.

Mendat, William:
Embedded Frontend-Development with .Net Core and Blazor / William Mendat. – BACHELOR THESIS, Offenburg: Offenburg University, 2022. 44 pages.

Vorwort

Die Entwicklung einer Benutzeroberfläche im Browser ist ein umfangreiches und sehr interessantes Thema, vorallem wenn die Möglichkeit gegeben ist, mit .Net Core im Browser zu programmieren. So ergibt sich die Möglichkeit eine Benutzeroberfläche für ein Non-Deeply Embedded System im Embedded Linux Bereich mit .Net Core zu Entwickeln.

Ich möchte mich an dieser Stelle bei allen Personen bedanken, die mich während dieser Bachelor-Thesis unterstützt haben. Ein besonderer Dank geht dabei an Prof. Dr.-Ing. Daniel Fischer für die Betreuung während der Arbeit und die Möglichkeit, dieses Thema überhaupt bearbeiten zu können. Des Weiteren möchte ich mich bei Phillip Schmidt und Lennard Donk bedanken, die mir während der Bearbeitung einige hilfreiche Tipps und Verbesserungsvorschläge gegeben haben.

Eidesstattliche Erklärung

Hiermit versichere ich eidesstattlich, dass die vorliegende Bachelor-Thesis von mir selbstständig und ohne unerlaubte fremde Hilfe angefertigt worden ist, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich oder dem Gedanken nach aus Veröffentlichungen, unveröffentlichten Unterlagen und Gesprächen entnommen worden sind, als solche an den entsprechenden Stellen innerhalb der Arbeit durch Zitate kenntlich gemacht habe, wobei in den Zitaten jeweils der Umfang der entnommenen Originalzitate kenntlich gemacht wurde. Ich bin mir bewusst, dass eine falsche Versicherung rechtliche Folgen haben wird.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Offenburg, 28.02.2022

William Mendat

Zusammenfassung

Embedded Frontend-Entwicklung mit .Net Core und Blazor

In dieser Abschlussarbeit wird eine .Net Core Blazor Anwendung für ein Non-Deeply Embedded System implementiert. Dabei werden verschiedene Ansätze auf dem Raspberry Pi implementiert, um einen aussagekräftigen Vergleich zwischen den hier verschiedenen Technologien zu schaffen. Ein Teil dieser Ausarbeitung besteht darin, eine Laufzeitanalyse zu erstellen.

Die Arbeit ermöglicht zunächst einen Einblick in den momentanen Stand der Technik, wie bislang eine GUI für ein Non-Deeply Embedded System implementiert wurde. Weitergehend soll veranschaulicht werden wie eine GUI mithilfe von .Net Core und Blazor implementiert werden könnte. Am Ende dieser Arbeit wird ein aussagekräftiges Fazit darüber abgegeben, ob die Technologie Blazor für die Frontend-Entwicklung im Non-Deeply Embedded Bereich geeignet ist.

Abstract

Embedded Frontend-Development with .Net Core and Blazor

In this thesis we implement a .Net Core Blazor Application for a non-deeply embedded system. Multiple approaches for a Raspberry Pi are implemented to create meaningful comparison between the different technologies. One part of this thesis is the creation of a run time analysis.

This thesis will first give an overview of the currently used technologies and how GUI's for non-deeply embedded system are created, before then showing how the same behaviour can be implemented in .Net Core using Blazor.

At the end of this thesis we conclude if the Blazor technology is suitable for use in front-end development for non-deeply embedded systems.

Inhaltsverzeichnis

1. Einleitung

Die Menschheit ist heutzutage darauf fokussiert, die komplette Welt zu digitalisieren. Dabei existiert ein Grundsatz:

> Alles, was digitalisiert werden kann, wird digitalisiert [1]

Um dies zu realisieren ist es vonnöten überall Hardware und Software zu verbinden. Sei es nun das Handy, mit welchem durch nur einen klick die Bankdaten angezeigt werden können, oder ein selbstfahrendes Auto, welches einen Anwender selbstständig zum Ziel fährt. Dies sind nur zwei Beispiele einer unendlich langen Liste. Hinter diesen technischen Wundern stecken meist mehrere Tausend kleiner Mikrocomputern und Mikrocontrollern, die dann mittels Software zusammen interagieren. Die Kombination dieser zwei Komponenten werden durch den Oberbegriff „Embedded System“ oder auch zu Deutsch ein „Eingebettetes System“ zusammengefasst.

Embedded Systems können dabei grundsätzlich zwischen zwei Plattformen unterschieden werden:

- Deeply Embedded System
- Non-Deeply Embedded System

Deeply Embedded Systems sind die wesentlichen Bausteine des Internet of Things [2]. Die Anwendungen, die bei Deeply Embedded Systems implementiert werden, basiert auf speziell angepassten Echtzeitbetriebssystemen. Zusätzlich werden die Programmiersprachen C und C++ sowie ganz spezielle GUI[1]-Frameworks wie zum Beispiel TouchGFX verwendet.

Anders als bei den Deeply Embedded Systems, die sehr auf speziellen Technologien aufbauen, bieten Non-Deeply Embedded Systems eine höhere Flexibilität in Sachen

[1] Graphical User Interface

Technologien an. Von Seiten der Programmierung ist es möglich, unterschiedliche Technologien und Programmiersprachen zu verwenden. Dort gilt bis dato unter Linux die Kombination von C++ und Qt für GUI-lastige Systeme als „State of the Art“.

Die Kombination aus C++ und Qt funktionierte bislang sehr gut. Jedoch kommt dieser Ansatz auch mit Herausforderungen. Nicht nur die höheren Entwicklungszeiten für die Entwicklung von C++ Anwendungen, sondern auch die geringe Verfügbarkeit von Experten auf dem Arbeitsmarkt sorgen für schlechtere Softwarequalität und längere Produktionszeiten.

Um diesen Herausforderungen zu entgegnen, soll in dieser Abschlussarbeit ein anderer Ansatz betrachtet werden. Und zwar könnte sowohl die Anwendungsschicht als auch die Persistenzschicht als .Net Core Anwendungen implementiert werden. Als GUI-Technologie soll dabei die neue Microsoft-Technologie namens *Blazor* als Qt-Ersatz zum Einsatz kommen. Somit kann erreicht werden, dass die komplette Applikation in .Net Core implementiert werden kann.

1.1. Aufgabenstellung

Das Ziel dieser Arbeit ist die Entwicklung eines Blazor-basierten Frontends auf einem Raspberry Pi 4B. Dabei soll ein aussagekräftiger Vergleich zwischen den Technologien geschaffen werden, um eine mögliche Verdrängung mittels Blazor zu demonstrieren. Dazu soll zunächst veranschaulicht werden, wie der momentane Stand der Technik für Non-Deeply Embedded Systems aussieht. Anschließend soll die Codebasis auf .Net Core und Blazor gewechselt werden. Insbesondere sollen dabei verschiedene Aspekte, wie zum Beispiel das Verhalten zur Laufzeit, dieses Ansatzes überprüft werden.

1.2. Verwendete Hardware

Als Zielplattform für diese Arbeit dient ein Raspberry Pi 4B. Der Raspberry Pi 4B verfügt dabei unter anderem über die folgenden technischen Spezifikationen: [3]

- 1,5 GHz ARM Cortex-A72 Quad-Core-CPU

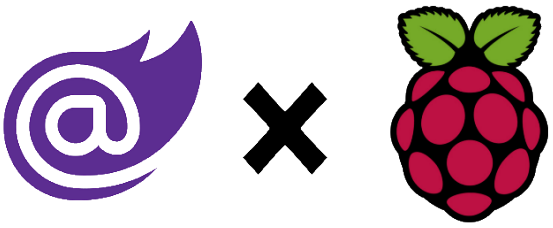

Abbildung 1.1: Blazor mit Raspberry Pi

- 4 GB LPDDR4 SDRAM
- Gigabit LAN RJ45 (bis zu 1000 Mbit)
- Bluetooth 5.0
- 2x USB 2.0 / 2x USB 3.0
- 2x microHDMI
- 5V/3A @ USB Typ-C
- 40 GPIO Pins
- Mikro SD-Karten Slot

Der Raspberry Pi 4B wird mit dem *Raspbian Buster with desktop* auf der SD-Karte betrieben.
Um noch mehr Funktionalität aufbringen zu können, wurde auf dem Raspberry Pi ein *RPI SENSE HAT Shield* aufgesteckt. Mithilfe des *RPI SENSE HAT Shield* können dann unter anderem Daten, wie zum Beispiel die momentane Temperatur, oder auch die momentane Luftfeuchtigkeit gewonnen werden. Zudem ist auf dem Raspberry Pi noch eine 8x8 LED-Matrix enthalten und ein Joystick mit 5 Knöpfen, die eingelesen werden können.

1.3. Verwendete Software

Im Rahmen dieser Arbeit werden die folgenden Softwaretools zur Entwicklung eingesetzt:

- Um eine Visualisierung des Images *Raspbian Buster with desktop* von dem Raspberry Pi zu erhalten, wird die Windows Desktop Anwendung *VNC View-*

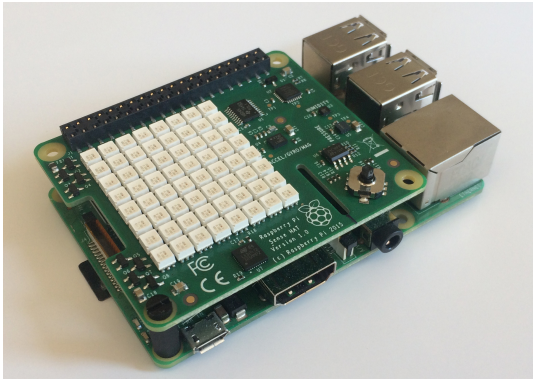

Abbildung 1.2: Verwendeter Raspberry Pi

er verwendet. Dadurch ist die Möglichkeit gegeben, bequem und einfach den Raspberry Pi Remote zu bedienen.

- Für die Demonstrierung des Kapitels *Stand der Technik* wird eine beispielhafte grafische Oberfläche mithilfe des Qt-Creators implementiert.
- Den Zugriff auf die Funktionalitäten des *RPI SENSE HAT Shield* wird mittels der *RTIMUlib* Bibliothek realisiert.
- Die Programmiersprachen, die Hauptsächlich in dieser Arbeit verwendet werden, sind C++ und C# beziehungsweise .Net Core.
- Um den Zugriff auf die Funktionalitäten des *RPI SENSE HAT Shield* mittels .Net Core zu gewährleisten, wird die IoT Bibliothek von Microsoft verwendet.
- Die Implementierung der Anwendung mit Blazor wird mittels der kostenlosen IDE *Visual Studio Code* realisiert.
- Um zwischen dem Host Rechner und dem Raspberry Pi Dokumente und Ordner auszutauschen, wird die Desktop-Applikation *WinSCP* verwendet.

Die oben vorgestellten Tools und Programmiersprachen wurden nicht explizit vorgegeben. Dementsprechend sind selbst ausgesucht und sind zu Beginn dieser Arbeit schon alle komplett eingerichtet.

2. Stand der Technik

Dieses Kapitel soll ein grundlegendes Verständnis wiedergeben, wie der momentane Stand der Technik aussieht. Dabei soll zunächst betrachtet werden, wie die derzeitige Entwicklung bei Non-Deeply Embedded Systems aussieht, um anschließend eine beispielhafte Anwendung zu implementieren.

2.1. Embedded Systems

Ein Embedded System oder auf Deutsch ein eingebettetes System wird als eine integrierte, mikroelektronische Steuerung angesehen. Welches meist darauf ausgelegt ist eine spezifische Aufgabe zu erledigen [4][vgl.]. Dabei setzt sich ein Embedded System aus dem Zusammenspiel zwischen Hardware und Software zusammen. Solche Embedded Systems haben meist kein ausgeprägtes Benutzerinterface und können weitergehend in die zwei Unterklassen, Non-Deeply und Deeply Embedded Systems unterteilt werden.

Neben der logischen Korrektheit, die Embedded Systeme erfüllen müssen, lassen sie sich durch eine Reihe unterschiedlicher Anforderungen und Eigenschaften von den heutzutage üblichen Anwedungen abgrenzen. Unter anderem werden bei Embedded Systems ein sogenanntes *Instant on* gefordert, welches besagt, dass das Gerät unmittelbar nach dem Einschalten betriebsbereit sein muss [5][vgl.].

Folgende Tabelle zeigt die typischen Anforderungen an ein Embedded System auf:

Anforderung	Beschreibung
Funktionalität	Die Software muss schnell und korrekt sein
Preis	Die Hardware darf nicht zu kostspielig sein
Robustheit	Muss auch in einem rauen Umfeld funktionieren
Fast poweroff	Muss in der Lage sein schnell das komplette System abzuschalten
Räumliche Ausmaße	Muss klein sein, um sich in ein System einbinden zu können
Nonstop-Betrieb	Muss in der Lage sein, im Dauerbetrieb laufen zu können
Lange Lebensdauer	Muss in der Lage sein, teilweise mehr als 30 Jahre zu laufen

Tabelle 2.1.: Anforderungen an Embedded Systems [5]

2.1.1. Hardware

Die einzelnen Komponenten, die in Embedded Systems verbaut worden sind, entscheiden über die vorhandene Leistung, den Stromverbrauch und die Robustheit, die das System ausmachen. Die Kernkomponente eines Embedded System wird durch einen Prozessor repräsentiert und wird häufig als *System on Chip* eingebaut. Dabei ist die Tendenz zu den ARM-Core Modellen steigend. Neben dem Prozessor befinden sich typischerweise auf den Embedded Systems weitere Komponenten, wie zum Beispiel dem Hauptspeicher, dem persistenten Speicher und diversen Schnittstellen, um weitere Peripheriegeräte anzuschließen [6][vgl.].

Damit weitere Peripheriegeräte angeschlossen werden können, müssen mittels einigen Leitungen digitale Signale übertragen werden. Aufgrund dessen, dass Leitungen typischerweise nur über eine begrenzte Leistung verfügen, werden Treiber eingesetzt, um peripherie Geräte zu verbinden. Nicht selten kommt es vor, dass in einem Embedded System darüber hinaus noch eine galvanische Entkopplung eingebaut wird. Dies dient dazu, die Hardware vor Störungen von außen, wie Beispiel Motoren, zu schützen [5][vgl.].

2.1.2. Software

Genauso wie sich Embedded Systems in zwei Bereiche unterscheiden lassen können, kann die Software eines Embedded System in Systemsoftware und funktionsbestimmende Anwendungssoftware unterteilt werden. Für ein *Deeply Embedded System* wird in den meisten fällen ein Echtzeitbetriebssystem (Realtime Operating

System - RTOS) verwendet, welches an die Hardware angepasst ist.

Ganz anders sieht es im *Non-Deeply Embedded Systems* Bereich aus. Da diese für sehr komplexe Aufgaben zum Einsatz kommen, kommt es nicht selten vor, dass eine GUI für ein solches System vonnöten ist. Deswegen basiert ein *Non-Deeply Embedded Systems* auf einer Systemsoftware, die Programmierer*innen mehr Möglichkeiten bei der Entwicklung geben. Unter anderem ist die Möglichkeit gegeben, die Programmiersprache und das GUI-Framework selbst auszusuchen [5][vgl.].

2.2. Qt

In der vorherigen Sektion wurde ein allgemeines Bild dargestellt, was ein Embedded System ist und in welchen Varianten diese auftauchen. Weitergehend soll in dieser Sektion vorgestellt werden, wie üblicherweise im *Non-Deeply Embedded Systems* Bereich programmiert wurde.

Ein großer Anteil eines *Non-Deeply Embedded Systems* wird heutzutage durch die GUI repräsentiert. Die GUI sollte intuitiv und zuverlässig sein. Zudem ist es enorm wichtig, dass die GUI wenige Ressourcen verbraucht, schnell reagiert und einfach einzubinden ist. Um diese Anforderung zu erfüllen, wird bis dato *Qt* in Kombination mit der Programmiersprache *C++* verwendet [7][vgl.].

2.2.1. Was ist Qt?

Qt ist ein Framework zum Erzeugen von GUIs auf mehreren Betriebssystemen. Es wurde 1990 von *Haarvard Nord* und *Eirik Chambe-Eng* mit der Intention benutzerfreundliche GUIs mit C++ zu entwickeln. Der Name *Qt* entstand dadurch, dass Haarvard den Buchstaben *Q* in Emacs als sehr schön empfand. Zudem steht das *t* in Qt als Abkürzung für das englische Wort *Toolkit* [8][vgl.].

Weitergehend sollte mit Qt die Möglichkeit geschaffen werden, mit nur einer Codebasis alle Betriebssysteme abzudecken. Die Schwierigkeit bestand also darin, dasselbe Aussehen und die gleiche Funktionalität über die verschiedenen Betriebssysteme zu schaffen [9][vgl.].

Qt ist in C++ entwickelt worden und verwendet zusätzlich noch einen Compiler welcher *moc*[1] genannt wird, mit welchem C++ um weitere Elemente erweitert wird. Der daraus kompilierte Code folgt dem C++-Standard und ist somit mit jedem anderen C++ Compiler kompatibel. Obwohl Qt ursprünglich dafür gedacht war, rein auf C++ zu basieren, wurden im Laufe der Zeit mehrere Eweiterungen für Qt von der Community entwickelt. Somit kann Qt mit mehreren Programmiersprachen benutzen werden, wie zum Beispiel Python oder Java.

Abbildung 2.1: Qt Logo

2.2.2. Programmierbeispiel

Das Programmierbeispiel welches im Folgenden dargestellt wird, erzeugt ein Fenster mit dem Titel *Meine erste Qt App*. Zudem ein Label welches *Hello World* anzeigt und ein Button mit der Aufschrift *Exit*. Sobald der Button wird, schließt sich das Fenster.

```
#include "mainwindow.h"

int main(int argc, char *argv[])
{
    // Set the Application
    QApplication app(argc, argv);
    QWidget window;
    // Set fixed Width of the Window
    window.setFixedWidth(300);

    // Set the Title of the Window
    window.setWindowTitle("Meine erste Qt App");

    // Create a Label and align it to Center
    QLabel *lblHello = new QLabel("Hello World!");
```

[1] meta object compiler

```
16      lblHello->setAlignment(Qt::AlignCenter);
17
18      // Create a Button and connect it to close the Window
19      QPushButton *btnExit = new QPushButton("&Exit");
20      // Connect button with the App
21      QObject::connect(btnExit, SIGNAL(clicked()), &app, SLOT(quit()));
22
23      // Sowohl das Label als auch die Schaltfläche vertikal ausrichten
24      QVBoxLayout *layout = new QVBoxLayout;
25
26      // Add the Widgets
27      layout->addWidget(lblHello);
28      layout->addWidget(btnExit);
29      window.setLayout(layout);
30
31      window.show();
32      return app.exec();
33  }
```

Listing 2.1: Qt Hello World Sourcecode

Daraus ergibt sich das folgende Programm

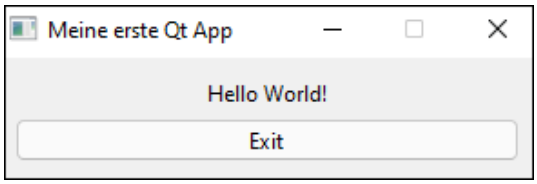

Abbildung 2.2: Qt Hello World App

2.2.3. Widgets

Zur Entwicklung einer GUI werden grafische Komponenten benötigt. Qt verwendet dafür sogenannte *Widgets*. Widgets sind grafische Komponenten um die Benutzeroberfläche zu gestalten. Ein Beispiel für eine solche Komponente wäre ein Button, welcher in der Sektion *Programmierbeispiel* als *btnExit* vorkam.

Die Widgets, die Qt zur Verfügung stellt sind in einer großen Klassenhierarchie zusammengesetzt und diese Hierarchie könnte sich wie folgt dargestellt werden:

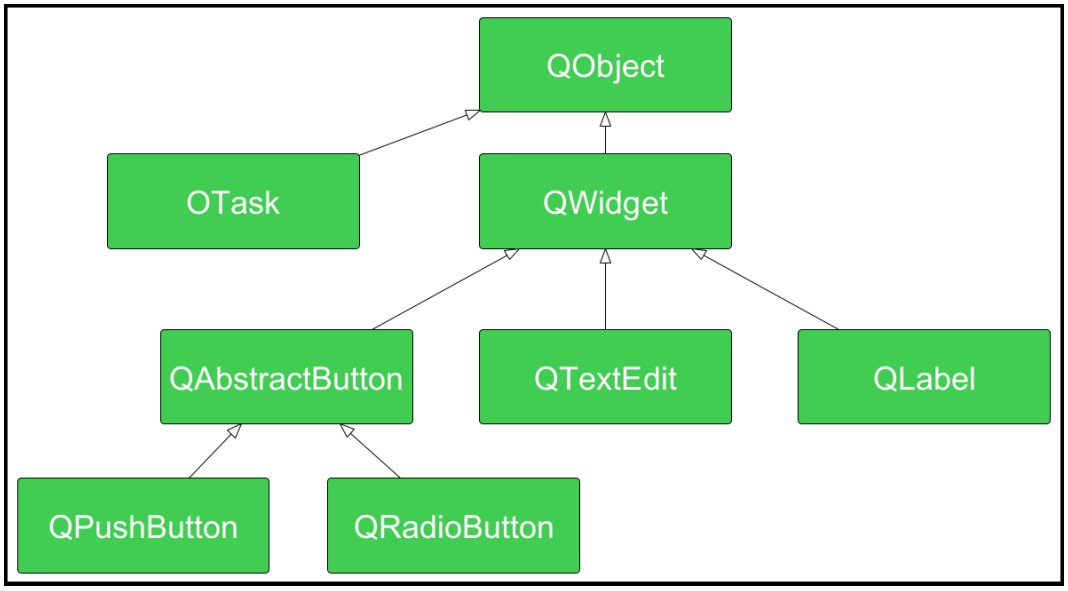

Abbildung 2.3: Qt Widgets Klassenhierarchie [9][vgl.]

Als Basisklasse ist ganz oben in der Klassenhierarchie die *QObject* Klasse. Diese enthält unter anderem den Signal- und Slot-Mechanismus. Auf diesen soll später noch genauer eingegangen werden. Weitergehend werden Widgets, die gemeinsame Funktionalitäten aufweisen zusammen gruppiert. Diese Verhalten ist bei *QPushButton* und *QRadioButton* erkennbar. Beide Widgets sind Buttons, die sich teilweise die gleichen Eigenschaften und Funktionen teilen [9][vgl.].

2.2.4. Signal und Slot Konzept

Eine interaktive Benutzeroberfläche muss auf Ereignisse reagieren können. Beispielsweise ist das Betätigen eines Buttons ein Auslöser für ein Ereignis auf welches reagiert werden kann. Es gibt viele Methoden und Muster in welche ein solches Ereignis-Aktionskonzept implementiert werden kann.

Ein *Signal* ist im einfachen Sinne eine Nachricht, die versendet werden kann. Die Nachricht signalisiert, dass sich der momentane Status eines Objektes geändert hat. Dahingegen ist ein *Slot* eine spezielle Funktion von einem Objekt, welche dann aufgerufen wird, wenn ein bestimmtes *Signal* gesendet wird [9][vgl.].

Damit jeder *Slot* auch weiß, auf welches *Signal* reagiert werden soll, müssen diese verbunden werden. In der Sektion *Programmierbeispiel* war folgende Zeile zu sehen:

```
1 // Connect button with the App
2 QObject::connect(btnExit, SIGNAL(clicked()), &app, SLOT(quit()));
```

Listing 2.2: Signal- und Slot-Beispiel

In dieser Codezeile fand die Verbindung zwischen einem Signal und einem Slot statt. Wenn der Button betätigt wird, wird *app* signalisiert und ruft *quit()* auf. Die Funktion *quit()* beendet das Programm.

Einer der größten Vorteile dieser Methode im Gegensatz zu anderen Methoden ist, dass dadurch n:m-Beziehungen abgebildet werden können. Das bedeutet also, dass sich ein Signal mit beliebig viel Slots verbinden kann und dass sich ein Slot auf mehrere Signale verbinden kann [10][vgl.].

2.2.5. Raspberry Pi und Qt

Nachdem ein grober Überblick zu Qt geschaffen wurde, soll eine beispielhafte Anwendung mithilfe von Qt auf dem Raspberry Pi implementiert werden. Bei dieser Anwendung soll beim Betätigen eines Buttons, Daten von dem Raspberry Pi gelesen werden und auf der GUI angezeigt werden. Dieses Beispiel ist von dem Embedded Systems 2 Labor inspiriert.

RTIMULib

Die RTIMU Bibliothek war anfändlich dazu gedacht, mithilfe von C++ oder Python Daten von einem Non-Deeply Embedded System zu lesen. Mittlerweile existiert diese Bibliothek auch für andere Sprachen wie zum Beispiel C#.

Mithilfe dieser Bibliothek soll für dieses Beispiel die Daten von dem Raspberry Pi gelesen werden, um diese anschließend auf der GUI anzeigen zu können. Um RTIMU in einem QT Projekt nutzen zu können, muss die Bibliothek zunächst in der *.pro* Datei eingebunden werden. Dies geschieht, indem die Zeile *LIBS += -lRTIMULib* hinzugefügt wird.

Hilfsklasse

Nachdem die Bibliothek hinzugefügt wurde, soll eine Hilfsklasse als Repräsentation der Daten erzeugt werden. Das folgende Listing zeigt dies auf:

```
class ReadData
{
private:
    float m_Temperature = 0.1f;
    float m_AirPressur = 0.1f;
    float m_Humidity = 0.1f;
    float m_xMagnetometer = 0.1f;
    float m_yMagnetometer = 0.1f;
    float m_zMagnetometer = 0.1f;

    RTIMUSettings* m_RTIMUSettings = nullptr;
    RTIMU* m_RTIMU = nullptr;
    RTPressure* m_RTPressure = nullptr;
    RTHumidity* m_RTHumidity = nullptr;

public:
    ReadData();
    void vRead(void);
};
```

Listing 2.3: RTIMU-Hilfsklasse

Die Klasse enthält neben den im Listing 2.3 gezeigten Code weitere Methoden, um auf die privaten *Member Variablen* zuzugreifen. Zudem müssen im Konstruktor die Variablen konfiguriert und initialisiert werden. Dies geschieht folgendermaßen:

```
ReadData::ReadData()
{
    // Define variables
    m_RTIMUSettings = new RTIMUSettings("RTIMULib");
    m_RTIMU = RTIMU::createIMU(pRTIMUSettings);
    m_RTPressure = RTPressure::createPressure(pRTIMUSettings);
    m_RTHumidity = RTHumidity::createHumidity(pRTIMUSettings);

    // Init
    pRTIMU->IMUInit();
    pRTIMU->setCompassEnable(true);
    pRTPressure->pressureInit();
    pRTHumidity->humidityInit();
}
```

Listing 2.4: RTIMU-Hilfsklasse-Konstruktor

Die letzte Komponente der Hilfsklasse, bildet die *vRead* Methode ab, in der schlussendlich die Daten gelesen werden:

```
void ReadData::vRead(void)
{
    if (m_RTIMU->IMURead())
```

```
    {
        RTIMU_DATA RTIMUData = m_RTIMU->getIMUData();
        m_RTPressure->pressureRead(RTIMUData);
        m_RTHumidity->humidityRead(RTIMUData);

        m_AirPressur = RTIMUData.pressure;
        m_Temperature = RTIMUData.temperature;
        m_fHumidity = RTIMUData.humidity;

        RTIMUData.compass.normalize();
        m_xMagnetometer = RTIMUData.compass.x();
        m_yMagnetometer = RTIMUData.compass.y();
        m_zMagnetometer = RTIMUData.compass.z();
    }
}
```

Listing 2.5: RTIMU-Hilfsklasse mit der Methode vRead

Benutzeroberfläche

Die Benutzeroberfläche in diesem Beispiel wurde schlicht gehalten. Sie besitzt insgesamt zwölf QtLabels, wovon sechs dafür gedacht sind, um die Raspberry Pi Daten abzubilden. Weitergehend enthält das Beispiel einen Button, der die Daten abrufen soll.

Der Button wurde mit einem Slot versehen der aufgerufen wird, wenn der Button betätigt wird. In dem Slot wird die vRead Methode aufgerufen, um dann die Labels zu aktualisieren. Die Implementierung des Slots sieht dann folgend aus:

```
void MainWindow::on_pbUpdate_clicked()
{
    m_readData->vRead();

    double dValue = static_cast<double>(m_readData->getAirPressur());
    ui->lblLuftdruck->setText(QString::number(dValue));

    dValue = static_cast<double>(m_readData->getTemperature());
    ui->lblTemperatur->setText(QString::number(dValue));

    dValue = static_cast<double>(m_readData->getHumidity());
    ui->lblLuftfreutigkeit->setText(QString::number(dValue));

    dValue = static_cast<double>(m_readData->getMagnetometerX());
    ui->lblKompassX->setText(QString::number(dValue));

    dValue = static_cast<double>(m_readData->getMagnetometerY());
    ui->lblKompassY->setText(QString::number(dValue));

    dValue = static_cast<double>(m_readData->getMagnetometerZ());
```

```
21     ui->lblKompassZ->setText(QString::number(dValue));
22 }
```

Listing 2.6: Slot Methode für den Update-Button

Das vollständige Programm ist in der folgenden Abbildung dargestellt:

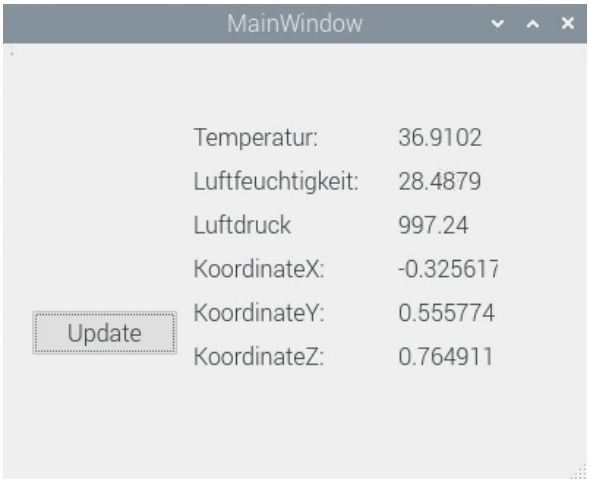

Abbildung 2.4: GUI der beispielhaften Anwendung

3. Blazor

Nachdem in der vorherigen Sektion das C++ Framework Qt vorgestellt wurde, soll in diesem Kapitel das Framework dieser Thesis vorgestellt werden, welches den Namen *Blazor* trägt.

3.1. Was ist Blazor?

Blazor ist ein Framework von Microsoft zum Erzeugen von Webseiten. Dabei macht Blazor gebrauch von den *Razor Pages*. Bei den *Razor Pages* handelt es sich um eine Technologie von Microsoft, die C# Elemente im *HTML-Markup* ermöglichen [11][vgl.]. Der Name *Blazor* entstand aus der Kombination der zwei Wörter *Browser* und *Razor*. Es wurde 2019 erstmalig von Microsoft veröffentlicht, mit der Intention Webseiten oder auch SPA[1] mithilfe von C# zu entwickeln. Dabei existieren zwei Varianten von Blazor:

- Blazor Server
- Blazor WebAssembly

Der essenzielle Unterschied der beiden Varianten besteht darin, dass Blazor Server auf einem Server gehostet wird und Blazor WebAssembly nativ im Browser läuft, dazu aber im späteren Verlauf dieser Arbeit mehr [12][vgl.].

Die Idee hinter *Blazor* ist, die Codebasis sowohl im Frontend als auch im Backend mit C# abzubilden. Somit wird erreicht, dass langjährige C#-Entwickler*innen mit ihrem vorhanden Wissen als Fullstack-Entwickler*innen eingesetzt werden können.

[1] Single Page Application

3.2. Architekturen

Da Blazor in zwei Varianten existiert, existieren dementsprechend zwei Architekturen für dieses Framework. Bevor die zwei Architekturen jedoch im Detail erklärt werden, sollen zuerst die Konzepte beschrieben werden, wie andere SPA wie beispielsweise Angular oder React funktionieren. Die heutigen SPA basieren auf der Client-Server-Architektur, das bedeutet der Client stellt eine Anfrage an den Server und erhält die passende Antwort. Die Kommunikation der beiden Teilnehmer geschieht in den meisten Fällen über eine REST API. Eine REST API ist im weitesten Sinne eine Programmierschnittstelle, die sich an den Paradigmen und Verhalten des World Wide Web (WWW) orientiert und einen Ansatz für die Kommunikation zwischen Client und Server in Netzwerken beschreibt [13][vgl.]. In dem folgenden Schaubild ist eine solche Architektur zu sehen:

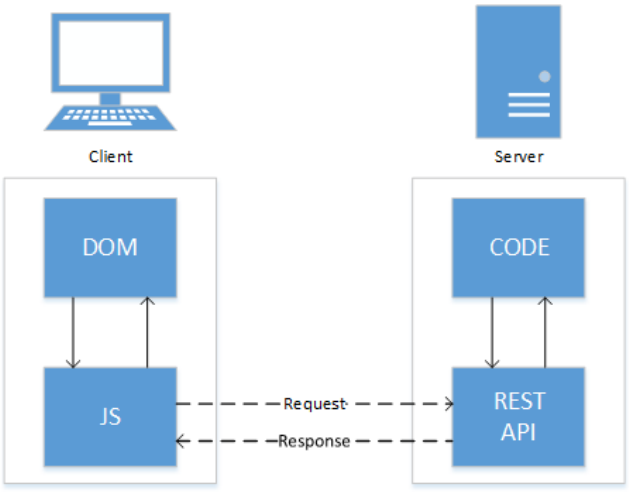

Abbildung 3.1: Client-Server Architektur mit Javascript

In Abbildung 3.1 ist aufgezeigt, wie der Client mithilfe von Javascript mit dem Server kommuniziert. Javascript holt sich die Daten, die der Client benötigt und gibt diese dem DOM zum Verarbeiten weiter.

3.2.1. Blazor WebAssembly

Die Architektur von Blazor WebAssembly ist ähnlich zu den obig beschriebenen SPAs. Tatsächlich verändert sich hierbei nur die Programmiersprache, die auf dem Client läuft. Es handelt sich dabei um die Programmiersprache C#, die auf dem Client ausgeführt wird, wie im Folgenden zu sehen ist.

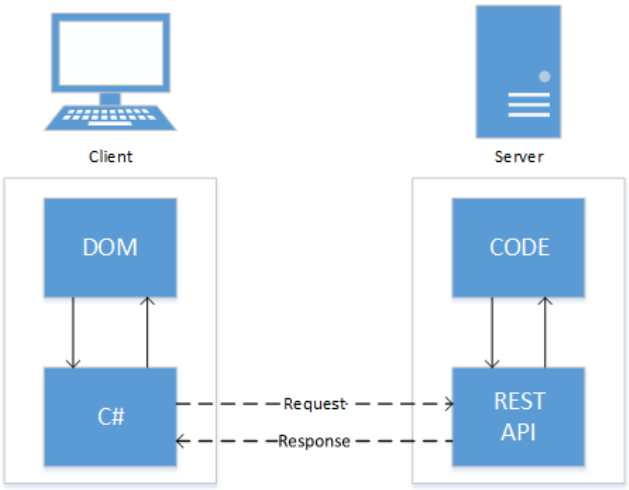

Abbildung 3.2: Blazor WebAssembly Architektur

Das ganze Konzept C# auf dem Client laufen lassen zu können, funktioniert durch WebAssembly. WebAssembly wandelt Programmcode in nativen Bytecode um, der in einer Sandbox im Browser ausgeführt werden kann. Dabei wird von der Sandbox aus die DOM mithilfe von Javascript kontinuierlich manipuliert. Javascript verschwindet in dem Sinne also nicht komplett, sondern wird lediglich ergänzt. Da für dieses Konzept also WebAssembly vonnöten ist, kann Blazor WebAssembly nicht auf Browsern funktionieren, die WebAssembly nicht unterstützen [14][vgl.].

Im Folgenden werden die Vor- und Nachteile von Blazor WebAssembly dargestellt:

+ Sehr skalierbar
+ Sehr performant
+ Es kann komplett eigenständig auf dem Client laufen und ist nicht auf den Server angewiesen
– Große Anwendungsdatei, die auf dem Client geladen werden muss
– Lange Ladezeit beim ersten Aufruf
– Kompletter Code ist auf dem Client sichtbar

3.2.2. Blazor Server

Anders als bei Blazor WebAssembly der Fall ist, wird bei Blazor Server nicht C# in den Browser geladen, sondern der Code wird Serverseitig ausgeführt. Deswegen wird auch die Web-Anwendung als SPA auf dem Server gerendert. Zum Client wird nur das JavaScript und Markup gesendet. Die Daten und Benutzereingaben werden laufend mittels Signal R zwischen Client und Server ausgetauscht. Dementsprechend ist es notwendig, dass immer zwischen dem Client und dem Server eine offene Verbindung vorhanden ist [14][vgl.].

Dadurch das die komplette Seite auf dem Server gerendert wird, lädt die Seite auf dem Client sehr schnell. Zudem muss lediglich eine kleine Javascript Datei auf den Client laden und die Seite kann auf leistungsschwachen Clients verwendet werden.

Das ganze Konzept dieser Architektur baut drauf auf, dass beim ersten Aufruf der Seite eine Javascript Datei names *Blazor.js* auf dem Client geladen wird. Diese Datei kommuniziert mit dem DOM und baut die Verbindung zum Server auf. Sobald die Verbindung aufgebaut ist, werden kontinuierlich Nachrichten zwischen Client und Server ausgetauscht.

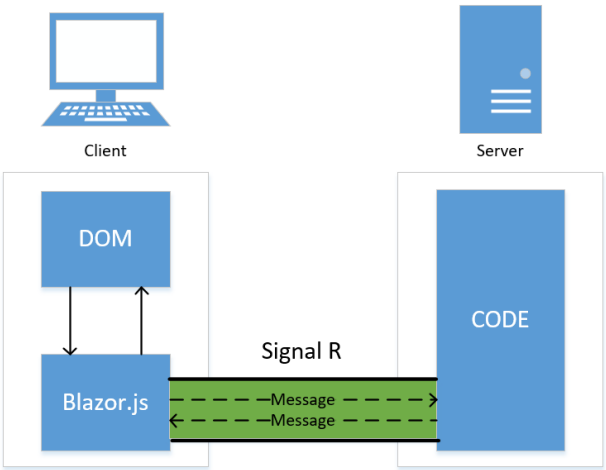

Abbildung 3.3: Serverarchitektur von Blazor

Im Folgenden werden die Vor- und Nachteile von Blazor Server zusammengestellt:

+ Kurze Ladezeiten
+ Komplett browserunabhängig
+ Client hat keinen Zugriff auf den Sourcecode

+ Ist in der Lage mit leistungsschwachen Clients zu interagieren
– Nicht skalierbar, da alle Benutzer auf einem Server zugreifen
– Lange Netzwerklatenz führt zu Verzögerungen in der Benutzeroberfläche
– Es muss immer eine Verbindung bestehen

3.3. Komponenten

Anders als bei Qt der Fall, basiert Blazor wie andere Web-Frameworks auf *HTML* und *CSS*. Das bedeutet, dass Entwickler*innen auf jedes HTML-Element zugreifen können. Weitergehend haben Entwickler*innen noch die Möglichkeit auf zusätzliche Komponentenanbieter wie zum Beispiel *MadBlazor* oder *Ignite UI* zurückzugreifen.

Außerdem bietet Blazor zusätzlich die Möglichkeit eigene Komponenten zu erstellen. Eine Komponente ist ein eigenständiger Teil der Benutzeroberfläche [15][vgl.]. Hinzukommend kann eine Blazor-Komponente, in zwei Varianten vorkommen. Einmal als eine *Page-Komponente* und eine *Non-Page-Komponente*.

- *Page-Komponente* kann über die URL adressiert werden
- *Non-Page-Komponente* kann nicht adressiert werden [15][vgl.]

```
@page "/counter"

<PageTitle>Counter</PageTitle>

<h1>Counter</h1>

<p role="status">Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

Listing 3.1: Beispiel einer Page-Komponente

Listing 3.1 zeigt eine Page-Komponete. Bei der Komponente handelt es sich um eine Page-Komponente, da diese mit *@page* in der ersten Zeile ausgezeichnet ist.

Einer Komponente kann auch ein oder mehrere Parameter von der Oberkomponente mitgegeben werden. Dies hat den Vorteil, dass Komponenten mehr an Flexibilität gewinnen und somit besser wiederzuverwenden sind. Wie im folgenden Beispiel zu sehen ist:

```
<h2>@Title</h2>

@code {
    [Parameter]
    public string Title { get; set; }
}
```

Listing 3.2: Kindkomponente

```
@page "/parent"

<h1>Parent</h1>

<Child Title="Child-Title"/>

@code {

}
```

Listing 3.3: Elternkomponente

Wie in Listing 3.2 zu sehen ist, bekommt die Kindkomponente ihren Title als Parameter übergeben. Listing 3.3 kann mithilfe des Title-Attributes den Title an die Kindkomponente übergeben. Der HTML-Tag der Kindkomponente, wird durch den Dateinamen der Komponente erzeugt. Da die Komponente *Child.razor* im Projekt heißt, wird diese mittels dem *<Child>-Tag* verwendet.

3.4. Javascript Interoperation

Durch Blazor ist die Möglichkeit gegeben, eine Webanwendung mit C# zu schreiben. Jedoch ist diese Technologie sehr neu und verfügt derzeit nicht über die Bandbreite an Funktionalitäten wie Javascript. Deswegen kann nicht pauschal gesagt werden, dass Blazor Javascript komplett ersetzt. Aufgrund dessen existiert in Blazor ein Mechanismus namens *Javascript Interoperation*. Dieser Mechanismus ermöglicht die Kommunikation zwischen C# und Javascript.

3.4.1. Javascript Runtime

Mithilfe des Javascript Runtime Interfaces, lassen sich Javascript Methoden aus C# Code aufrufen. In dem Interface befindet sich eine Methode names *InvokeAsync*, die jedoch in zwei Varianten zur Verfügung steht:

- ValueTask<TValue> InvokeAsync<TValue>(string, object?[]?)
- ValueTask<TValue> InvokeAsync<TValue>(string, CancellationToken, object?[]?)

Der zweiten Funktion kann ein *CancellationToken* mitgegeben werden, um die Methode manuell abzubrechen [16][vgl.].

Des Weiteren gilt für die Verwendung des Javascript Runtime Interface Folgendes:

- Der *string* Parameter in InvokeAsync steht für den Javascript Methodennamen.
- Der *object?[]?* Parameter ist ein Array von Objekten, um der Javascript Methode die Parameter zu übergeben.
- TValue ist ein Template-Objekt, welches als Rückgabewert benutzt werden kann. Zudem muss das Objekt mithilfe von *JSON* serialisierbar sein.
- Der Javascript Code muss unter dem Verzeichnis *wwwroot* vorhanden sein.
- Um diesem Mechanismus anzuwenden, muss in der Komponente das Interface *IJSRuntime injected* werden
- Bei Blazor Server können Javascript Methoden erst aufgerufen werden, sobald der Signal R Channel aufgebaut ist.

Um diesen Mechanismus zu demonstrieren, soll im Folgenden ein Beispiel dargestellt werden. In diesem Beispiel wird ein String in einer Javascript-Methode erzeugt und zurückgegeben. Der erzeugte String wird dann im C# Code abgefangen und auf der Benutzeroberfläche angezeigt.

```
function generateString(name) {
    return "Hello " + name + ", u have clicked the button!";
}
```

Listing 3.4: Javascript-Methode generateString

```
@page "/jsExample"
@inject IJSRuntime _js

<h1>Javascript Runtime</h1>

```

```
<p>@output</p>

<button @onclick="CallJS">Click Me</button>

@code {
    private string output = string.Empty;

    private async Task CallJS()
    {
        output = await _js.InvokeAsync<string>("generateString", "Mustermann");
    }
}
```

Listing 3.5: Beispiel einer Javascript-Komponente

3.4.2. Javascript Invokable

Nun kann es den Fall geben, dass C# Code von Javascript aufgerufen werden muss. Dies wird mit dem *JSInvokable*-Attribut möglich. Die Funktion, die mit dem *JSInvokable*-Attribut erweitert wurde, muss zudem mit dem *static-keyword* versehen werden.

Javascript-seitig werden von Blazor die Methoden

- DotNet.invokeMethod (string, string, object[])
- DotNet.invokeMethodAsync (string, string, object[])

bereitgestellt, mit denen eine statische C#-Methode aufgerufen werden kann. Dabei geben Entwickler*innen mithilfe des ersten Parameters, den Project Namen an, in der sich die statische Methode befindet. Der zweite Parameter, ist für den Namen der aufzurufenden Methode. Mit dem dritten Parameter können weitere optionale Argumente übergeben werden.

Im Folgenden wird eine Komponente gezeigt die eine Javascript-Methode aufruft. In der Javascript-Methode wird wiederum eine C#-Methode aufgerufen. Das Ergebniss der C#-Methode wird anschließen in einem *alert* angezeigt.

```
@page "/jsInvokable"
@inject IJSRuntime _js

<h1>Javascript Invokable</h1>

<button onclick="showGeneratedMessage()">Click Me</button>

@code {

    [JSInvokable]
    public static Task<string> GenerateString()
    {
        return Task.FromResult("This is called from Javascript");
    }

}
```

Listing 3.6: Javascript Invokable Beispiel

```
function showGeneratedMessage() {
    DotNet.invokeMethodAsync('Test', 'GenerateString')
        .then(data => {
            alert(data)
        });
}
```

Listing 3.7: Javascript Invokable showGeneratedMessage

3.5. Blazor Maui

Mit dem Release von .Net 6 wurde Blazor mit Maui kombiniert. Die Abkürzung *Maui* steht dabei für *Multi-platform Application UI*. Mit Blazor Maui sollen zukünftig native Desktop- oder Mobile Applikationen mit Blazor erstellt werden können.

Die Architektur von Blazor Maui sieht vor, dass der Blazor Code durch Maui in nativen Code umgewandelt wird. Dieser kann dadurch auf der jeweiligen Platform ausgeführt zu werden. Was im folgenden Bild zu erkennen ist:

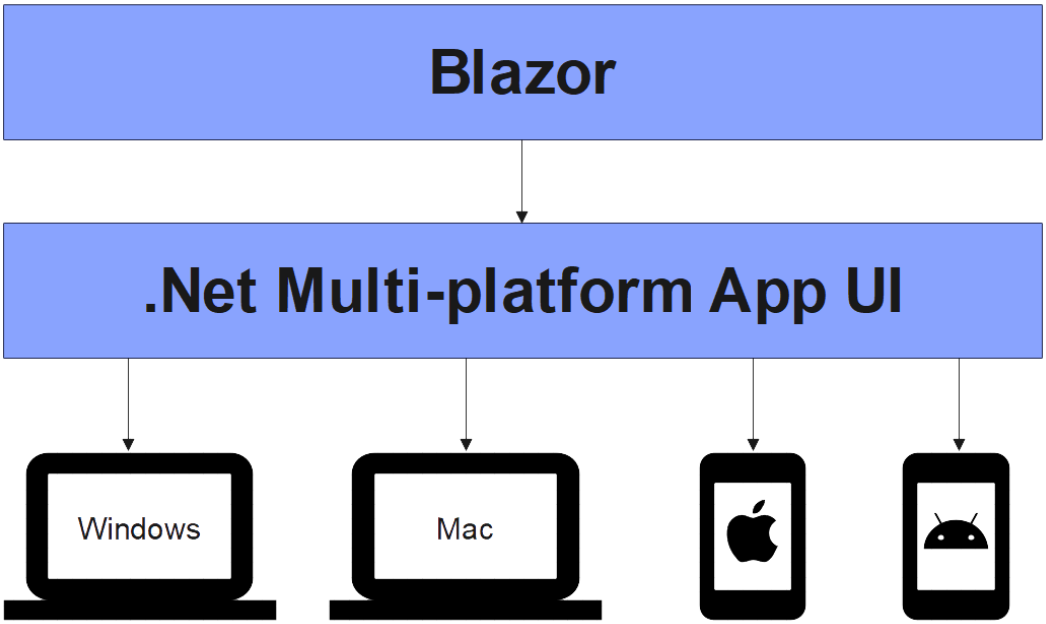

Abbildung 3.4: Blazor Maui Architektur

Wie zu sehen ist, werden Windows, Mac, iOS und Android mithilfe von Maui untersützt. Linux hingegen bleibt zu dem Zeitpunkt dieser Arbeit noch außen vor.

Aufgrund dessen, das Linux zum derweiligen Zeitpunkt noch nicht unterstützt wird, und Non-Deeply Embedded System meist auf Linux basieren, wird Blazor Maui nicht weitergehend in dieser Arbeit behandelt.

4. Raspberry Pi mit .Net Core und Blazor

In diesem Kapitel soll ein einfaches Non-Deeply Embedded System mithilfe von Blazor auf einem Raspberry Pi 4B erstellt werden. Dabei soll gezeigt werden, welche Entwicklungsumgebung genutzt werden kann und was auf dem Target installiert werden muss.

4.1. Entwicklungsumgebung

Die Entwicklungsumgebung, die in dieser Arbeit eingesetzt wird, ist Visual Studio Code. Dabei wurde Visual Studio Code deswegen ausgesucht, da viele Community Plugins zur Verfügung stehen, die das Entwickeln auf dem Raspberry Pi unterstützten. Zum einen ist die Möglichkeit gegeben, sich Remote mit den Raspberry Pi zu verbinden und zum anderen existiert die Möglichkeit des Remote Debuggings. Beim Remote Debugging wird der Code auf dem Host Rechner entwickelt und auf dem Target ausgeführt.

Im Zuge dieser Arbeit wird sich Remote mit dem Raspberry Pi verbunden, um dann auf dem Target zu programmieren. Dafür muss die Extension *Remote - SSH* von Microsoft in Visual Studio Code installiert werden. Nun kann mit der Tastenkombination *Strg - Shift - p* ein neuer Dialog geöffnet werden, in welchem die Option *Remote-SSH Connect to host* ausgewählt werden kann. Nachdem *Remote-SSH Connect to host* ausgewählt wurde, muss der Befehl *<benutzernamen>@<ip adresse>* eingegeben werden. Zuletzt muss lediglich das Password eingegeben werden.

4.2. Installation

Da in der vorherigen Sektion *Entwicklungsumgebung* Visual Studio Code eingerichtet wurde und mit dem Raspberry Pi Remote verbunden wurde, kann das integrierte Terminal von Visual Studio Code für die Installation von .Net Core verwendet werden.

Im ersten Schritt muss überprüft werden, ob es sich bei dem Raspberry Pi um die 32-bit oder die 64-bit Version handelt. Dies kann mit dem Befehl *uname -a* überprüft werden. Dabei können folgende zwei Ergebnisse erfolgen:

```
Linux raspberrypi 4.19.97-v7l+ 1294 SMP Thu Jan 30 13:23:13 GMT 2020
armv7l GNU/Linux
Linux raspberrypi 5.10.60-v8+ 1291 SMP Thu Jan 30 13:21:14 GMT 2020
aarch64 GNU/Linux
```

Beim ersten Ergebnis handelt es sich um 32-Bit Version und beim zweiten um die 64-Bit Version.

Die jeweilige SDK kann entweder auf der offiziellen Microsoftseite oder in einem Terminal mit dem Befehl *wget* heruntergeladen werden. Nachdem der Download beendet ist, kann mit den folgenden Befehlen die Installation beendet werden:

```
mkdir -p $HOME/dotnet
tar zxf dotnet-sdk-6.0.100-rc.1.21458.32-linux-arm.tar.gz -C $HOME/dotnet
export DOTNET_ROOT=$HOME/dotnet
export PATH=$PATH: $HOME/dotnet
```

Diese Befehle erstellen einen neuen Ordner namens *dotnet*, entpacken die SDK und packen den Inhalt in den neu erstellten Ordner. Zudem wird die *Path-Variable* zu dem dotnet Ordner.

Mit dem Befehl *dotnet –info* kann anschließend überprüft werden, ob die Installation erfolgreich war.

4.3. Blazor Demo Anwendung

Nachdem in der letzten Sektion *Installation* .Net Core installiert wurde, soll in dieser Sektion eine beispielhafte Blazor Anwendung auf dem Raspberry Pi erstellt werden. Die Anwendung wird auf der *Blazor Server* Architektur basieren. Dabei sollen kontinuierlich Daten von dem Raspberry Pi abgefragt und auf der Benutzeroberfläche abgebildet werden.

4.3.1. Erstellen des Projektes

Um die Anwendung zu erstellen muss der folgende Befehl im Teminal eingegeben werden:

```
dotnet new blazorserver -o MyApp –no-https
```

Dieser Befehl erstellt eine neue Blazor Server Anwendung mit dem Namen *MyApp* und konfiguriert die Anwendung ohne das HTTPS-Protokol. Der Name sowie dass kein HTTPS konfiguriert wird sind optionale Parameter, die nicht mit angegeben werden müssen. Nachdem die Anwendung erfolgreich erstellt wurde, muss die Codezeile *webBuilder.UseUrls("Http://*:5000");* in der *Program.cs*-Datei angegeben werden. Damit wird sichergestellt, dass alle Geräte im LAN darauf zugreiffen können. Der Code in der *Program.cs*-Datei sieht dann folgendermaßen aus:

```
1  public class Program
2  {
3      // Main
4
5      public static IHostBuilder CreateHostBuilder(string[] args) =>
6          Host.CreateDefaultBuilder(args)
7              .ConfigureWebHostDefaults(webBuilder =>
8              {
9                  webBuilder.UseStartup<Startup>();
10                 webBuilder.UseUrls("Http://*:5000"); // <----
11             });
12 }
```

Listing 4.1: Program.cs Code

Mit dem Befehl *dotnet run* kann die Anwendung gestartet werden. Sobald das Programm gestartet ist, kann mit dem Link *http://<ip>:5000* die Seite erreicht werden. Die aufgerufene Seite sollte wie folgt angezeigt werden:

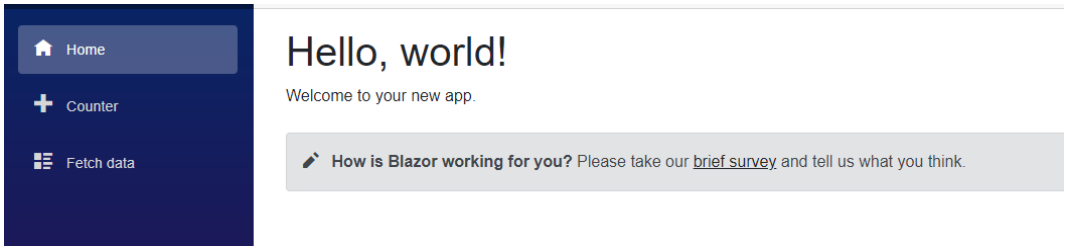

Abbildung 4.1: Blazor Server Template

4.3.2. Microsoft IoT

Damit mit dem Raspberry Pi kommuniziert werden kann, können Bibliotheken verwendet werden. Microsoft bietet eine IoT Bibliothek an, mit der die Sensoren oder die LEDs auf dem Gerät angesteuert werden können. Die Bibliothek kann wie folgt in das Projekt eingebunden werden:

```
1    <ItemGroup>
2    <PackageReference Include="Iot.Device.Bindings" Version="1.5.0-*" />
3  </ItemGroup>
```

Listing 4.2: IoT NuGet Package

Das folgende beispielhafte Konsolenprogramm demonstriert, wie auf die Daten zugegriffen werden kann:

```
1    public class Program
2    {
3        static void Main(string[] args)
4        {
5            using SenseHat _senseHat = new();
6
7            Console.WriteLine($"Temperatur: {_senseHat.Temperature.DegreesCelsius
                 :0.#}\u00B0C");
8            Console.WriteLine($"Temperatur 2: {_senseHat.Temperature2.DegreesCelsius
                 :0.#}\u00B0C");
9            Console.WriteLine($"Luftdruck: {_senseHat.Pressure.Hectopascals:0.##}
                 hPa");
10           Console.WriteLine($"Luftfeuchtigkeit: {_senseHat.Humidity.Percent:0.#}%"
                 );
11       }
12   }
```

Listing 4.3: SenseHat Beispielprogramm

Der Output durch obiges Programm sieht wie folgt aus:

Temperatur: 38,4°C
Temperatur 2: 38,5°C
Luftdruck: 984,03 hPa
Luftfeuchtigkeit: 31%

4.3.3. Raspberry Pi Daten anzeigen

In dieser Sektion soll die Logik implementiert werden, um die Daten vom Raspberry Pi auf der Benutzeroberfläche anzuzeigen. Dafür wird ein *Timer* implementiert, der die Daten ausliest.

Damit etwas auf der Benutzeroberfläche angezeigt werden kann, muss das *HTML-Markup* implementiert werden:

```
<div class="divHeader">
    <div>
        <h1>Raspberry Pi</h1>

        <h2>Uhrzeit: @_uhrzeit</h2>
    </div>

    <div>
        <h3>Temperature Sensor 1: @_temperatur</h3>
        <h3>Temperature Sensor 2: @_temperatur2</h3>
        <h3>Luftdruck: @_pressure</h3>
        <h3>Luftfeuchtigkeit: @_humidity</h3>
    </div>
</div>
```

Listing 4.4: HTML-Markup

Dabei signalisiert das @<*name*>, dass es sich um eine Variable handelt, die im Code deklariert wurde.

Weitergehend bietet Blazor verschiedene *Render-Funktionen* die nach bestimmten Ereignissen aufgerufen werden. Wie zum Beispiel die *OnInitializedAsync*, die aufgerufen wird, sobald die Komponente geladen wird. Diese *OnInitializedAsync* kann dazu genutzt werden, um die Variablen zu initialisieren.

```
    protected override Task OnInitializedAsync()
    {
        _senseHat = new();
        _cultureInfo = new("de-DE");
        _uhrzeit = DateTime.Now.ToString("HH:mm:ss", _cultureInfo);

```

```
7         _temperatur = string.Empty;
8         _temperatur2 = string.Empty;
9         _pressure = string.Empty;
10        _humidity = string.Empty;
11
12        return base.OnInitializedAsync();
13    }
```

Listing 4.5: Render-Funktion: OnInitializedAsync

Es soll zudem eine Hilfsfunktion *SetRaspValues* geschaffen werden, die die Daten ausliest. Diese Funktion wird in der *OnInitializedAsync* Funktion aufgerufen.

```
1     private void SetRaspValues()
2     {
3         _temperatur = $"{_senseHat.Temperature.DegreesCelsius:0.#}\u00B0C";
4         _temperatur2 = $"{_senseHat.Temperature2.DegreesCelsius:0.#}\u00B0C";
5         _pressure = $"{_senseHat.Pressure.Hectopascals:0.##} hPa";
6         _humidity = $"{_senseHat.Humidity.Percent:0.#}%";
7     }
```

Listing 4.6: Funktion: SetRaspValues

Die Hilfsfunktion kann in einem Timer genutzt werden. Dieser Timer soll jede Sekunde aufgerufen werden, um die Werte zu aktualisieren.

```
1     private void StartTimer()
2     {
3         _readTimer = new(1000);
4         _readTimer.Elapsed += GetData;
5         _readTimer.Enabled = true;
6     }
7
8     private void GetData(Object source, System.Timers.ElapsedEventArgs e)
9     {
10        _uhrzeit = DateTime.Now.ToString("HH:mm:ss", cultureInfo);
11        SetRaspValues();
12        InvokeAsync(StateHasChanged);
13    }
```

Listing 4.7: Timer: $_readTimer$

Da die Werte in einem Timer aktualisiert werden, kann *Blazor* die Änderungen nicht automatisch feststellen. Dieses Verhalten kommt dadurch, da der Timer nicht auf dem *Ui-Thread* läuft. Um *Blazor* zu signalisieren, dass Änderungen vorhanden sind, kann die Funktion *StateHasChanged* aufgerufen werden. Diese Funktion löst einen neuen *Render-Vorgang* aus.

Der momentane Stand der Seite sieht so aus:

Abbildung 4.2: Zwischenstand der Blazor Demo

4.3.4. LED-Matrix ansteuern

In dieser Sektion soll eine *8x8 Button-Matrix* erstellt werden, die die *8x8 LED-Matrix* auf dem Sensehat des Raspberry Pi repräsentieren soll. Dabei soll der vorhandene Joystick auf dem Sensehat genutzt werden, um über die Button-Matrix zu navigieren und die LEDs zu platzieren.

Da Blazor die *Razor-Syntax* verwendet, kann jegliche C# Kontrollstruktur im *HTML-Markup* verwendet werden. So kann dies wie folgt genutzt werden, um die *8x8 Button-Matrix* zu erzeugen:

```
<div class="divGrid">
    @for (var y = 0; y < LengthY; y++)
    {
        @for (var x = 0; x < LengthX; x++)
        {
            int copyY = y;
            int copyX = x;
            <Button class="buttonBox @classes[copyY,copyX]"/>
        }
    }
</div>
```

Listing 4.8: Button-Matrix

Das *@classes* Element ist ein Zwei-Dimensionales Array aus Strings, mit der CSS-Klassen hinzugefügt und wieder gelöscht werden sollen. Um zu ermitteln, welcher Joystick-Button betätigt wurde, soll ein weiterer *Timer* zum Einsatz kommen. Dieser Timer soll alle 15 Millisekunden ausgelöst werden.

```
    private void StartTimer()
    {
        // More Code

        _setButtonTimer = new(15);
        _setButtonTimer.Elapsed += WriteStateToChannel;
        _setButtonTimer.Enabled = true;
    }


    private async void WriteStateToChannel(Object source, System.Timers.
        ElapsedEventArgs e)
    {
```

```
13          if((ticks - lastTicks) > 9){
14              _senseHat.ReadJoystickState();
15              if(_senseHat.HoldingButton){
16                  await _stateChannel.Writer.WriteAsync(JoystickState.Holding);
17              }
18              else if(_senseHat.HoldingUp){
19                  await _stateChannel.Writer.WriteAsync(JoystickState.Up);
20              }
21              else if(_senseHat.HoldingDown){
22                  await _stateChannel.Writer.WriteAsync(JoystickState.Down);
23              }
24              else if(_senseHat.HoldingLeft){
25                  await _stateChannel.Writer.WriteAsync(JoystickState.Left);
26              }
27              else if(_senseHat.HoldingRight){
28                  await _stateChannel.Writer.WriteAsync(JoystickState.Right);
29              }
30              lastTicks = ticks;
31          }
32          ticks++;
33      }
```

Listing 4.9: Timer: $_setButtonTimer$

Bei *JoystickState* handelt es sich um ein Enum, welches den *State* des Joysticks repräsentieren soll. Wie zu sehen ist, wird der aktuelle *State* des Joysticks gelesen, um das Ergebnis in einen *Channel* zu schreiben.

Für die Verarbeitung des *States*, wird in *OnInitializedAsync* ein Task gestartet, der in einer Endlosschleife läuft und darauf warte bis etwas in den *Channel* hinzugefügt wird. Sobald sich der *State* geändert hat, wird entweder die Position berechnet oder die LED an dieser Position leuchtet in einer neuen Farbe.

```
1           _setButtonTask = Task.Run(async () =>
2           {
3               while (true)
4               {
5                   if(cancellationToken.IsCancellationRequested)
6                       cancellationToken.ThrowIfCancellationRequested();
7                   var state = await _stateChannel.Reader.ReadAsync();
8                   int x = 0;
9                   int y = 0;
10                  if(state == JoystickState.Holding){
11                      SetButtonBackground(_currentX, _currentY);
12                  }
13                  else if(state == JoystickState.Up){
14                      y--;
15                  }
16                  else if(state == JoystickState.Down){
17                      y++;
18                  }
19                  else if(state == JoystickState.Left){
```

```
                x--;
            }else if(state == JoystickState.Right){
                x++;
            }
            SetPositions(x, y);
            RemoveButtonBorder(_previousX, _previousY);
            SetButtonBorder(_currentX, _currentY);
            await InvokeAsync(StateHasChanged);
        }
    });

private void SetButtonBackground(int x, int y)
{
    if (classes[y, x].Contains(" setBackground"))
    {
        _senseHat.SetPixel(x, y, Color.Blue);
        classes[y, x] = classes[y, x].Replace(" setBackground", string.Empty);
    }
    else
    {
        _senseHat.SetPixel(x, y, Color.Red);
        classes[y, x] += " setBackground";
    }
}
```

Listing 4.10: Task zum Verarbeiten des States

Wie zu erkennen ist, wird die LED und der Button je nach vorherigem Zustand entweder Blau oder Rot. Die entstandene Anwendung sieht dann wie folgt aus:

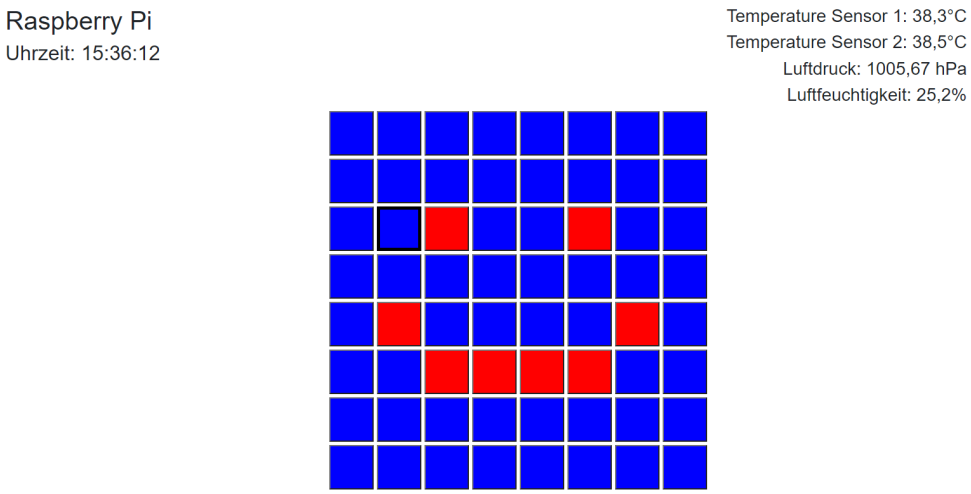

Abbildung 4.3: Blazor Demo

Da nur technisch relevante Code Abschnitte in diesem Kapitel präsentiert wurden, ist der komplette Code im Anhang A.1 zu finden.

5. Analyse

Dieses Kapitel befasst sich mit der Analyse der beiden vorgestellen Frameworks. Dabei werden mithilfe der beiden Blazor Architekturen und Qt verschieden Szenarien implementiert und die benötigte Ausführungszeit dokumentiert.

Um eine Analyse zwischen verschiedenen GUI-Frameworks durchzuführen, gibt es mehrere Ansätze. In diesem Kapitel wird auf dem Ansatz zurückgegriffen, der in [17] aufgeführt ist. Dabei werden folgende Methodiken abgedeckt.

- Analyse Bedingungen
- Analyse Metriken
- Analyse Szenarien
- Zusammenfassung der Analyse

Die Szenarien werden sich in zwei Teile aufteilen. Zuerst werden kleinere Codesegmente aufgelistet und danach größere Datenstrukturen analysiert und ausgewertet.

5.1. Analyse Bedingungen

Für die Analyse, die für diese Arbeit durchgeführt wurde, gelten die folgenden Bedingungen:

- Die Default Konfigurationen und Einstellungen für die IDE wurden benutzt.
- Alle Szenarien wurden auf der selben Hardware implementiert und ausgeführt.
- Alle Benutzerprogramme, außer die eigentliche IDE und ein zusätlicher Browser, wurden für die Auswertung geschlossen.

- Der vorgestellte Blazor Code wurde für Blazor Server und Blazor WebAssembly genutzt.
- Der Blazor Code wurde lediglich lokal getestet, um faire Bedingungen zu erschaffen.
- Jedes Szenario, sowohl die kleinen Codesegmente als auch die größeren Datenstrukturen, wurde 1000 mal ausgeführt und der Durchschnittswert betrachtet.
- Lediglich wurde die Zeit für die Ausführungs des gezeigten Codes gemessen.

5.2. Analyse Metriken

Für die Analyse wird die Ausführungzeit als Maßstab genommen. Die Ergebnisse der Ausführungszeiten werden gegenüber gestellt und verglichen. Dabei werden die Szenarien, die weniger Ausführungszeit in Anspruch genommen haben, als besser angesehen.

Als Beispiel, sei ein Szenario *A* (geschrieben in Blazor Server) gegeben und die Ausführungzeit ist im Vergleich kürzer als das gleiche Szenario (geschrieben in Blazor Webassembly oder Qt). In diesem Fall würde das Szenario *A* als das performanteste angesehen werden.

5.3. Analyse Szenarien

Die Szenarien werden in mehrere Bereiche unterteilt. Zuerst werden kleinere Codesegmente sowohl in Blazor als auch in Qt aufgelistet. Danach werden größere Datenstrukturen mit einer variablen Anzahl an Daten befühlt. Dabei werden die folgenden zwei Datenstrukturen verwendet:

- Tabelle
- Baum

5.3.1. Codesegmente

Im Folgenden werden 9 Szenarien aufgeführt. Da sich jedoch die Technologien Blazor und Qt stark unterscheiden, konnte nicht immer die gleiche Struktur angewendet werden. So kommt es dazu, dass hierbei auf das gleiche Verhalten geachtet wurde, als auf die Programmierstruktur. Aus diesem Grund unterscheiden sich die folgenden Szenarien sehr, führen jedoch zum gleichen Ergebnis.

Szenario #1
Erzeugen einer neuen leeren Combobox auf der Benutzeroberfläche.

Blazor Code:
```
string widget = @«select></select>";
myMarkup = new(widget);
```

Qt Code:
```
QComboBox* widget = new QComboBox();
ui->verticalLayout->addWidget(widget);
```

Szenario #2
Erzeugen eines neuen Labels auf der Benutzeroberfläche.

Blazor Code:
```
string widget = @«label>Text</lable>";
myMarkup = new(widget);
```

Qt Code:
```
QLabel* widget = new QLabel("Text");
ui->verticalLayout->addWidget(widget);
```

Szenario #3
Erzeugen eines neuen Buttons auf der Benutzeroberfläche.

Blazor Code:
```
string widget = @«button>Text</button>";
myMarkup = new(widget);
```

Qt Code:
QPushButton* widget = new QPushButton("Text");
ui->verticalLayout->addWidget(widget);

Szenario #4

Erzeugen einer neuen Checkbox auf der Benutzeroberfläche.

Blazor Code:
string widget = @«input type='checkbox'></input>";
myMarkup = new(widget);

Qt Code:
QCheckBox* widget = new QCheckBox();
ui->verticalLayout->addWidget(widget);

Szenario #5

Erzeugen einer neuen Textbox auf der Benutzeroberfläche.

Blazor Code:
string widget = @«input type='text'></input>";
myMarkup = new(widget);

Qt Code:
QTextEdit* widget = new QTextEdit();
ui->verticalLayout->addWidget(widget);

Szenario #6

Erzeugen einer neuen Combobox mit fünf Elementen auf der Benutzeroberfläche.

Blazor Code:
string widget = @«select>
<option value='0'>Text1</option>
<option value='1'>Text2</option>
<option value='2'>Text3</option>
<option value='3'>Text4</option>
<option value='4'>Text5</option>
</select>";
myMarkup = new(widget);

Qt Code:

```
QComboBox* widget = new QComboBox();
widget->insertItem(0, QString::fromStdString("Text1"));
widget->insertItem(1, QString::fromStdString("Text2"));
widget->insertItem(2, QString::fromStdString("Text3"));
widget->insertItem(3, QString::fromStdString("Text4"));
widget->insertItem(4, QString::fromStdString("Text5"));
ui->verticalLayout->addWidget(widget);
```

Szenario #7

Den Text von einer Textbox lesen.

Blazor Code:

```
string textToRead = string.Empty;
<input type="text"@bind-value="@textToRead/>
var text = textToRead;
```

Qt Code:

```
auto text = ui->textEdit->toPlainText();
```

Szenario #8

Den Text eines Labels verändern.

Blazor Code:

```
string textToWrite = string.Empty;
<label>@textToWrite</label>
textToWrite = "Text";
```

Qt Code:

```
ui->lblTextlabel->setText(QString::fromStdString("Text"));
```

Szenario #9

Den Text einer Textbox verändern.

Blazor Code:

```
string textToWrite = string.Empty;
<input type="text"@bind-value="@textToWrite/>
textToWrite = "Text";
```

Qt Code:

```
ui->textEdit->setText(QString::fromStdString("Text"));
```

Von den oben gegebenen Codesegmenten resultieren folgende Ergebnisse. Die Ergebnisse werden in Mikrosekunden angegeben.

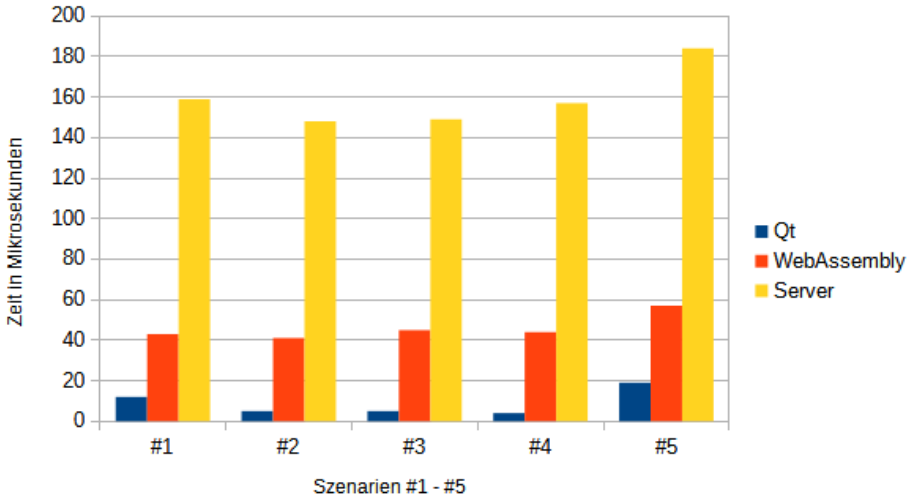

Abbildung 5.1: Ergebnisse der Szenarien #1 - #5 in Microsekunden

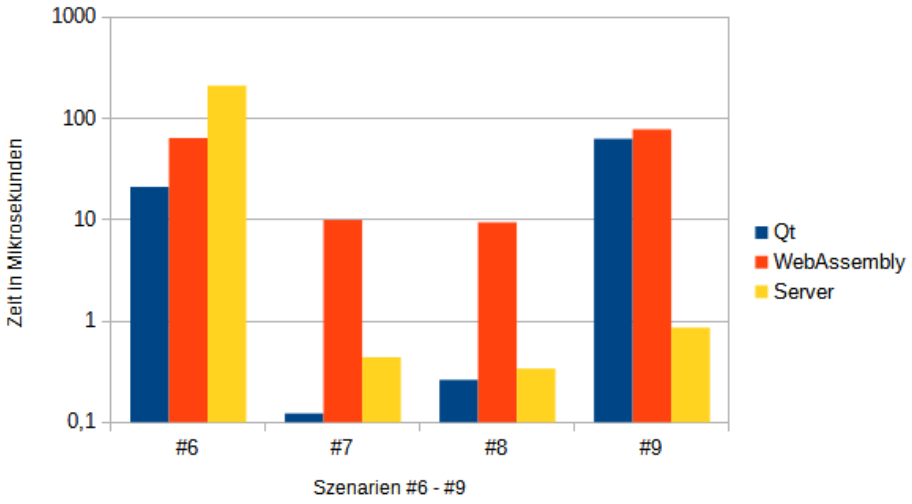

Abbildung 5.2: Ergebnisse der Szenarien 6 - 9 in Microsekunden

5.3.2. Tabelle

Die Tabelle ist einer der meist genutzten Datenstrukturen um Daten zu repräsentieren. Aus diesem Grund wurde die Tabelle in der Analyse mit aufgenommen. Dabei wurde eine Tabelle von *N* Elementen generiert und die Ergebnisse der Messungen im folgenden Diagram zusammengetragen:

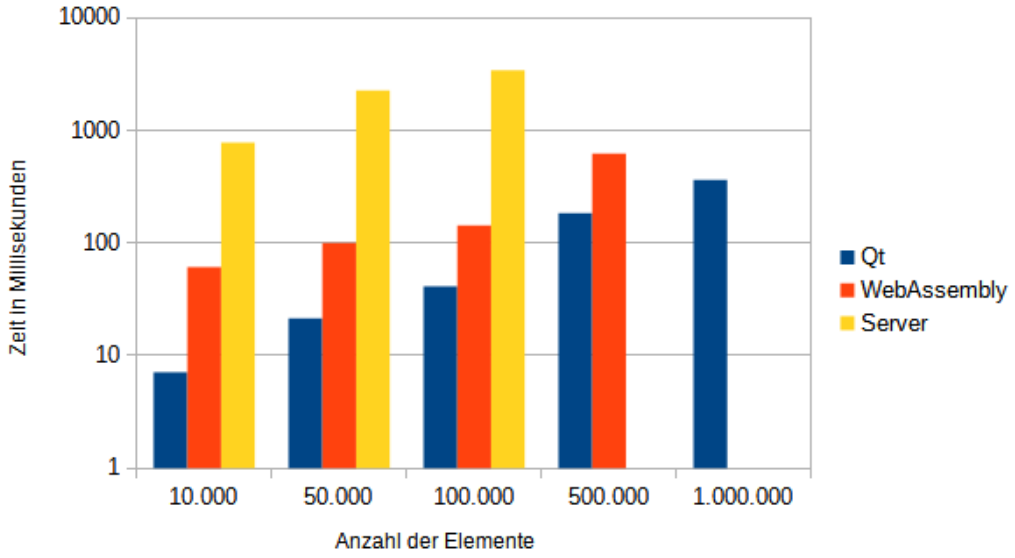

Abbildung 5.3: Ergebnisse der Messungen von der Erzeugung einer Tabelle in Millisekunden

Interessant ist zu erkennen, dass die Server-Architektur bei 500.000 Elementen in einem *Timeout* resultierte. Dieses Vehalten lässt sich dadurch erklären, da die Elemete vom Server zum Client übertragen werden müssen.

Die WebAssembly-Architektur wiederum, resultierte erst bei 1.000.000 Elementen in einem *Timeout*. Hier mussten die Elemente nicht zuerst vom Server zum Client übertragen werden, sondern standen direkt zum Verarbeiten auf dem Client zur Verfügung. Der *Timeout* bei 1.000 .000 Elementen, lässt sich dadurch erklären, dass der *Rendering-Prozess* bei vielen Elementen zu lange dauert.

5.3.3. Baumstruktur

Die Baumstruktur ist dafür gedacht, Elemente zu repräsentieren, die wiederum Unterelemente enthalten können. Ein klassisches Beispiel ist dabei ein *Filesystem*. Es existieren in einem *Filesystem* sowohl *Ordner* als auch *Dateien*. Ein *Odner* kann

sowohl mehrere *Dateien* als auch weitere *Ordner* in sich tragen. In den weiteren *Ordnern* können sich wiederum Elemente befinden.

In diesem Szenario wurden jeweils *NxM* Bäume erzeugt. Dabei steht *N* für die Anzahl der übergeordneten Elementen und *M* für die Anzahl der untergeordneten Elementen. Die Ergebnisse der Messungen resultierten in das folgende Diagram:

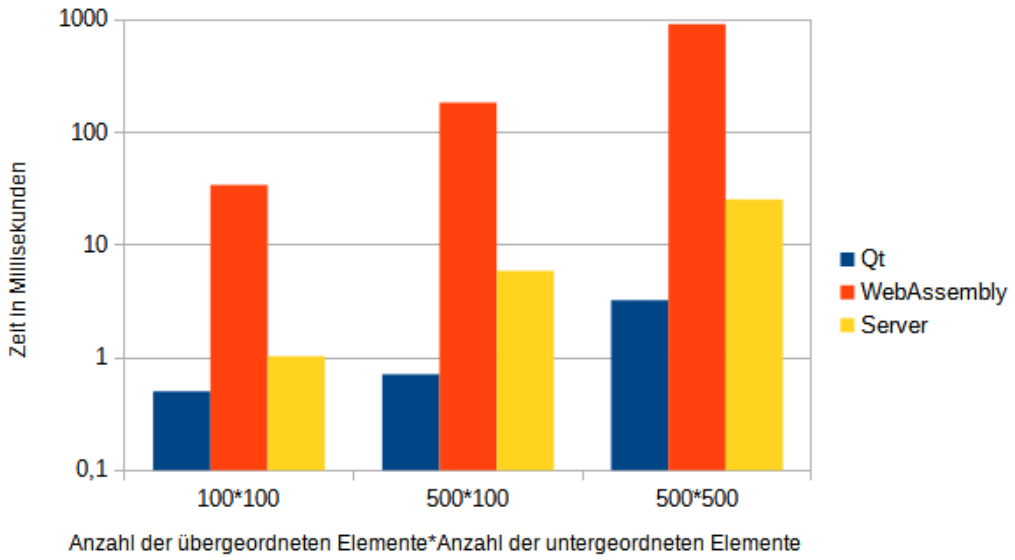

Abbildung 5.4: Ergebnisse der Messungen von der Erzeugung einer Baumstruktur in Millisekunden

In diesem Diagram ist zu erkennen, das die Server-Architektur performanter als die WebAssembly-Architektur ist. Dies könnte dadurch erklärt werden, da mit wesentlich weniger Elementen gearbeitet wurde als in der Sektion *Tabelle*. Dadurch müssen nicht so viele Elemente vom Server zum Client übertragen werden.

5.3.4. Zusammenfassung

Das primäre Ziel dieser Analyse war es, die Performance von den zwei GUI-Frameworks zu testen. Die Ergebnisse dieser Analyse zeigten, dass Qt Blazor in fast allen vorgestellten Szenarien überlegen war. Lediglich beim Szenario 9 *Den Text einer Textbox verändern* zeigte sich, dass die Blazor Server-Architektur leicht überlegen war.

Ein möglicher Grund für die Überlegenheit von Qt könnte sein, dass Qt eine native Desktopanwendung ist. Die Überlegenheit von Qt könnte sich auch dadurch erklären, da C++ eine sehr hardwarenahe Programmiersprache ist.

Interessant war jedoch auch das Verhalten der beiden Blazor-Architekturen zu beobachten. Werden die ersten sechs vorgestellten Szenarien betrachtet zeigt sich, dass die WebAssembly-Architektur wesentlich schneller als die Server-Architektur scheint. Dies könnte daran liegen, dass die Server-Architektur, die generierten Elemente erst zum Client senden muss.

Anders scheint es bei den Szenarien sieben bis neun, bei denen etwas gelesen oder geschrieben wurde. Dort zeigt sich die Server-Architektur als wesentlich performanter. Hier könnte die Überlegenheit daran liegen, da die Server-Architektur, die Daten schon lokal zur Verfügung stehen. Somit können die Daten direkt verarbeitet werden.

Werden die Ergebnisse der Tabellenstruktur und der Baumstruktur gegenübergestellt, so könnte behauptet werden, dass die Server-Architektur performanter scheint, wenn mit kleineren Datenmengen gearbeitet wird. Die WebAssembly-Architektur hingegen, zeigte sich mit größeren Datenmengen performanter.

6. Fazit

In dieser Arbeit wurde veranschaulicht, wie das Framework *Blazor* genutzt werden kann, um eine Benutzeroberfläche für ein Non-Deeply Embedded System zu erstellen. Es konnte gezeigt werden, welche Bibliotheken benutzt werden können, um auf die Funktionalitäten eines Non-Deeply Embedded System zuzugreifen. In dem entwickelten Frontend, wurden kontinuierlich Daten vom Raspberry Pi abgefragt, um diese auf der Benutzeroberfläche anzuzeigen. Im Anhang A.1 befindet sich der komplett implementierte Code.

Zusätzlich wurde im Umfang dieser Arbeit eine Analyse zwischen Qt und den beiden Blazor-Architekturen durchgeführt. Bei dieser Analyse stellte sich herraus, dass Qt in den meisten Fällen am performantesten war. Zudem zeigte sich ein interessantes Verhalten zwischen den beiden Blazor-Architekturen. Bei geringen Datenmengen, scheint die Server-Architektur performanter als die WebAssembly-Architektur zu sein. Jedoch bei großen Datenmengen, ist die WebAssembly-Architektur deutlich besser als die Server-Architektur.

7. Ausblick

Abschließend soll noch ein kurzer Ausblick über die weitere Entwicklung von Benutzeroberflächen für Non-Deeply Embedded Systems gegeben werden. Die beiden vorgestellten Frameworks dieser Arbeit eignen sich beide für die Entwicklung einer Benutzeroberfläche. Jedoch muss hier differenziert werden, welche Kriterien die Benutzeroberfläche erfüllen muss.

In dem Fall, dass die Performance sehr wichtig ist, sollte auf das Framework Qt in Kombination mit C++ zurückgegriffen werden. Sollte jedoch ein schneller Entwicklungsprozess vonnöten sein, ist das Framework Blazor definitiv eine gute Alternative.

In dieser Arbeit wurde zudem noch die Blazor Maui-Architektur vorgestellt, die aufgrund der nicht vorhandenen Kompatibilität mit Linux, nicht weiter betrachtet werden konnte. Hier bleibt es abzuwarten, ob Microsoft Linux mit *Blazor* unterstützten wird. Sollte Linux in Zukunft von Blazor Maui profitieren, so sollte eine solche Analyse erneut durchgeführt werden.

Abkürzungsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literatur

[1] *Alles, was digitalisiert werden kann, wird digitalisiert – Digitalisierung und ich*, 5.02.2022. Adresse: https://digitalisierung-und-ich.de/alles-was-digitalisiert-werden-kann-wird-digitalisiert/.

[2] Hochschule niederrheim, *CAS Embedded Systems Professional - Hochschule Niederrhein*, 29.10.2021. Adresse: https://www.hs-niederrhein.de/weiterbildung/sichere-software/cas-embedded-systems-professional/.

[3] *Raspberry Pi 4 Model B specifications – Raspberry Pi*, 5.11.2021. Adresse: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

[4] S. Gerstl, „Was ist ein Embedded System?“, *Embedded Software Engineering*, 27.11.2017. Adresse: https://www.embedded-software-engineering.de/was-ist-ein-embedded-system-a-665424/.

[5] J. Quade, *Embedded Linux lernen mit dem Raspberry Pi: Linux-Systeme selber bauen und programmieren*, 1. Auflage. Heidelberg: dpunkt.verlag, 2014, ISBN: 9783864915093. Adresse: http://ebooks.ciando.com/book/index.cfm/bok_id/1423957.

[6] Whatis.com/de, *Was ist Embedded System (Eingebettetes System)? - Definition von WhatIs.com*, 14.02.2022. Adresse: https://whatis.techtarget.com/de/definition/Eingebettetes-System.

[7] Q. Jinhui, L. D. Hui und Y. Junchao, „The Application of Qt/Embedded on Embedded Linux“, in *2012 International Conference on Industrial Control and Electronics Engineering*, 2012, S. 1304–1307. DOI: 10.1109/ICICEE.2012.346.

[8] *the-qt-story*, 22.03.2016. Adresse: https://rtime.felk.cvut.cz/osp/prednasky/gui/the-qt-story/.

[9] B. Baka, *Getting Started with Qt 5: Introduction to programming Qt 5 for cross-platform application development*, 1. Aufl. Birmingham: Packt

Publishing Limited, 2019, ISBN: 9781789955125. Adresse: https://www.wiso-net.de/document/PKEB__9781789955125136.

[10] M. Lobur, I. Dykhta, R. Golovatsky und J. Wrobel, „The usage of signals and slots mechanism for custom software development in case of incomplete information“, in *2011 11th International Conference The Experience of Designing and Application of CAD Systems in Microelectronics (CADSM)*, 2011, S. 226–227.

[11] Rick-Anderson, *Einführung in Razor Pages in ASP.NET Core*, 17.02.2022.

[12] Kexugit, *Web Development - C# in the Browser with Blazor*, 3.12.2021. Adresse: https://docs.microsoft.com/en-us/archive/msdn-magazine/2018/september/web-development-csharp-in-the-browser-with-blazor.

[13] D. Srocke, „Was ist eine REST API?“, *CloudComputing-Insider*, 9.06.2017. Adresse: https://www.cloudcomputing-insider.de/was-ist-eine-rest-api-a-611116/.

[14] bbv, *Hier kommt Blazor*, 4.12.2021. Adresse: https://www.bbv.ch/blazor/.

[15] Guardrex, *ASP.NET Core-Razor-Komponenten*, 17.02.2022. Adresse: https://docs.microsoft.com/de-de/aspnet/core/blazor/components/?view=aspnetcore-6.0.

[16] ——, *Aufrufen von JavaScript-Funktionen über .NET-Methoden in ASP.NET Core Blazor*, 10.12.2021. Adresse: https://docs.microsoft.com/de-de/aspnet/core/blazor/javascript-interoperability/call-javascript-from-dotnet?view=aspnetcore-6.0.

[17] H. B. Hassan und Q. I. Sarhan, „Performance Evaluation of Graphical User Interfaces in Java and C“, in *2020 International Conference on Computer Science and Software Engineering (CSASE)*, 2020, S. 290–295. DOI: 10.1109/CSASE48920.2020.9142075.

A. Anhang

```
@page "/"
@implements IAsyncDisposable
@using System.Drawing;
@using System.Threading
@using System.Threading.Channels
@using System.Globalization
@using Iot.Device.Common;
@using Iot.Device.SenseHat;

<div class="divHeader">
    <div>
        <h1>Raspberry Pi</h1>

        <h2>Uhrzeit: @_uhrzeit</h2>
    </div>

    <div>
        <h3>Temperature Sensor 1: @_temperatur</h3>
        <h3>Temperature Sensor 2: @_temperatur2</h3>
        <h3>Luftdruck: @_pressure</h3>
        <h3>Luftfeuchtigkeit: @_humidity</h3>
    </div>
</div>

<div class="divGrid">
    @for (var y = 0; y < LengthY; y++)
    {
        @for (var x = 0; x < LengthX; x++)
        {
            int copyY = y;
            int copyX = x;
            <Button class="buttonBox @classes[copyY,copyX]" @onclick="() =>
                SetButtonBackground(copyX, copyY)" />
        }
    }
</div>

@code {

    private enum JoystickState {
        EMTPY = 0,
```

```
41          Holding,
42          Up,
43          Down,
44          Right,
45          Left,
46      }
47
48      private const int LengthY = 8;
49      private const int LengthX = 8;
50
51      private string _uhrzeit;
52
53      private System.Timers.Timer _timeTimer;
54      private System.Timers.Timer _setButtonTimer;
55      private Random _randomNumberGenerator;
56
57      private CultureInfo cultureInfo;
58
59      private Channel<JoystickState> _stateChannel;
60      private Task _setButtonTask;
61      private SenseHat _senseHat;
62
63      private CancellationTokenSource _tokenSource;
64
65      string _temperatur = string.Empty;
66      string _temperatur2 = string.Empty;
67      string _pressure = string.Empty;
68      string _humidity = string.Empty;
69
70      int _previousX;
71      int _previousY;
72
73      int _currentX;
74      int _currentY;
75
76      string[,] classes = new string[LengthY, LengthX];
77
78      uint ticks = 0;
79      uint lastTicks = 0;
80
81      protected override Task OnInitializedAsync()
82      {
83          _stateChannel = Channel.CreateUnbounded<JoystickState>();
84          _randomNumberGenerator = new();
85          cultureInfo = new("de-DE");
86          _uhrzeit = DateTime.Now.ToString("HH:mm:ss", cultureInfo);
87          _senseHat = new();
88          _tokenSource = new();
89          var cancellationToken = _tokenSource.Token;
90
91          for (var y = 0; y < LengthY; y++)
92          {
93              for (var x = 0; x < LengthX; x++)
```

```
94              {
95                  classes[y, x] = string.Empty;
96              }
97          }
98
99          _senseHat.Fill(Color.Blue);
100
101         StartTimer();
102         InitPositions();
103         SetRaspValues();
104
105         SetButtonBorder(_currentX, _currentY);
106
107         _setButtonTask = Task.Run(async () =>
108         {
109             while (true)
110             {
111                 if(cancellationToken.IsCancellationRequested)
112                     cancellationToken.ThrowIfCancellationRequested();
113
114                 var state = await _stateChannel.Reader.ReadAsync();
115
116                 int x = 0;
117                 int y = 0;
118
119                 if(state == JoystickState.Holding){
120                     SetButtonBackground(_currentX, _currentY);
121                 }
122                 else if(state == JoystickState.Up){
123                     y--;
124                 }
125                 else if(state == JoystickState.Down){
126                     y++;
127                 }
128                 else if(state == JoystickState.Left){
129                     x--;
130                 }else if(state == JoystickState.Right){
131                     x++;
132                 }
133
134                 SetPositions(x, y);
135
136                 RemoveButtonBorder(_previousX, _previousY);
137                 SetButtonBorder(_currentX, _currentY);
138
139                 await InvokeAsync(StateHasChanged);
140             }
141         });
142
143         return base.OnInitializedAsync();
144     }
145
146     private void StartTimer()
```

```
147     {
148         // Every Second
149         _timeTimer = new(1000);
150         _timeTimer.Elapsed += GetData;
151         _timeTimer.Enabled = true;
152 
153         // Every 15 Milliseconds
154         _setButtonTimer = new(15);
155         _setButtonTimer.Elapsed += WriteStateToChannel;
156         _setButtonTimer.Enabled = true;
157     }
158 
159     private void InitPositions(){
160         _previousX = 0;
161         _previousY = 0;
162         _currentX = 0;
163         _currentY = 0;
164     }
165 
166     private void SetPositions(int newX, int newY){
167         _previousX = _currentX;
168         _previousY = _currentY;
169 
170         _currentX = (_currentX + LengthX + newX) % 8;
171         _currentY = (_currentY + LengthY + newY) % 8;
172     }
173     private void GetData(Object source, System.Timers.ElapsedEventArgs e)
174     {
175         _uhrzeit = DateTime.Now.ToString("HH:mm:ss", cultureInfo);
176 
177         SetRaspValues();
178 
179         InvokeAsync(StateHasChanged);
180     }
181 
182     private async void WriteStateToChannel(Object source, System.Timers.
        ElapsedEventArgs e)
183     {
184         if((ticks - lastTicks) > 9){
185 
186             _senseHat.ReadJoystickState();
187 
188             if(_senseHat.HoldingButton){
189                 await _stateChannel.Writer.WriteAsync(JoystickState.Holding);
190             }
191             else if(_senseHat.HoldingUp){
192                 await _stateChannel.Writer.WriteAsync(JoystickState.Up);
193             }
194             else if(_senseHat.HoldingDown){
195                 await _stateChannel.Writer.WriteAsync(JoystickState.Down);
196             }
197             else if(_senseHat.HoldingLeft){
198                 await _stateChannel.Writer.WriteAsync(JoystickState.Left);
```

```
199             }
200             else if(_senseHat.HoldingRight){
201                 await _stateChannel.Writer.WriteAsync(JoystickState.Right);
202             }
203 
204             lastTicks = ticks;
205         }
206         ticks++;
207     }
208 
209     private void SetRaspValues()
210     {
211         _temperatur = $"{_senseHat.Temperature.DegreesCelsius:0.#}\u00B0C";
212         _temperatur2 = $"{_senseHat.Temperature2.DegreesCelsius:0.#}\u00B0C";
213         _pressure = $"{_senseHat.Pressure.Hectopascals:0.##} hPa";
214         _humidity = $"{_senseHat.Humidity.Percent:0.#}%";
215     }
216 
217     public async ValueTask DisposeAsync()
218     {
219         _timeTimer.Dispose();
220         _setButtonTimer.Dispose();
221         _tokenSource.Cancel();
222         _tokenSource.Dispose();
223         _senseHat.Dispose();
224     }
225 
226     private void SetButtonBackground(int x, int y)
227     {
228         if (classes[y, x].Contains(" setBackground"))
229         {
230             _senseHat.SetPixel(x, y, Color.Blue);
231             classes[y, x] = classes[y, x].Replace(" setBackground", string.Empty);
232         }
233         else
234         {
235             _senseHat.SetPixel(x, y, Color.Red);
236             classes[y, x] += " setBackground";
237         }
238     }
239 
240     private void RemoveButtonBorder(int x, int y){
241         if (classes[y, x].Contains(" setBorder"))
242         {
243             classes[y, x] = classes[y, x].Replace(" setBorder", string.Empty);
244         }
245     }
246 
247     private void SetButtonBorder(int x, int y){
248         if (!classes[y, x].Contains(" setBorder"))
249         {
250             classes[y, x] += " setBorder";
251         }
```

```
252     }
253
254 }
255
256 Css:
257 .setBorder {
258     border: solid 5px black;
259 }
260
261 .setBackground {
262     background-color: red !important;
263 }
264
265 .buttonBox {
266     width: 100%;
267     height: 100%;
268     background: blue;
269 }
270
271 .divGrid {
272     display: grid;
273     grid-template-columns: repeat(8, 70px);
274     grid-template-rows: repeat(8, 70px);
275     grid-gap: 5px;
276     justify-content: center
277 }
278
279 .divHeader {
280     display: flex;
281     flex-direction: row;
282     justify-content: space-between;
283 }
284
285 .divHeader div {
286     display: flex;
287     flex-direction: column;
288 }
289
290     .divHeader div:last-child {
291         align-items: flex-end;
292         margin-right: 100px;
293     }
294
295         .divHeader div:last-child h3 {
296             text-align: right;
297         }
```

Listing A.1: Kompletter Demo Code