

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 Offen-
burg, 2022. 22 Seiten.

Mendat, William:

Embedded Frontend-Development with .Net Core and Blazor / William Mendat. –
BACHELOR THESIS, Offenburg: Offenburg University, 2022. 22 pages.

Vorwort

—...—

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 implementiert, um somit einen Ersatz für die derzeit GUI-Entwicklung auf dem Raspberry Pi mit Qt zu kreieren. Dabei werden verschiedene Ansätze implementiert, sowohl auf dem Raspberry Pi als auch auf einem externen Server, um einen aussagekräftigen Vergleich zwischen den hier verschiedenen Technologien zu schaffen. Ein großer Teil dieser Ausarbeitung besteht darin, eine Laufzeitanalyse zu erstellen.

Die Arbeit gibt zunächst einen Einblick in den momentanen Stand der Technik, wie bislang mit einem Raspberry Pi gearbeitet wurde, um dann zu veranschaulichen, wie dass gleiche Verhalten mit Net Core und Blazor widergespiegelt werden kann. Am Ende dieser Arbeit wird ein aussagekräftiges Fazit darüber abgegeben, ob die Technologie Blazor für die Frontendentwicklung im Embedded Bereich zu gebrauchen ist.

Abstract

Embedded Frontend-Development with .Net Core and Blazor

Englische Version von Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Aufgabenstellung	2
1.2. Verwendet Hardware	3
1.3. Verwendete Software	4
2. Stand der Technik	5
2.1. Embedded Systems	5
2.1.1. Hardware	6
2.1.2. Software	7
2.2. Qt	7
2.2.1. Was ist Qt?	8
2.2.2. Programmierbeispiel	9
2.2.3. Widgets	10
2.2.4. Signal und Slot Konzept	11
2.2.5. Raspberry Pi und Qt	11
3. Blazor	12
3.1. Was ist Blazor?	12
3.2. Architekturen	13
3.2.1. Blazor WebAssembly	13
3.2.2. Blazor Server	15
3.3. Komponenten	16
3.4. Javascript Interoperation	18
3.5. Blazor MAUI	18
4. Raspberry Pi mit .Net Core und Blazor	19
4.1. Entwicklungsumgebung	19
4.2. Installation	19
4.3. Programmieren mit .Net Core	19
4.4. Blazor Demo Anwendung	19
5. Analyse	20
6. Fazit	21

7. Ausblick	22
Abkürzungsverzeichnis	
Tabellenverzeichnis	i
Abbildungsverzeichnis	ii
Quellcodeverzeichnis	iii
Literatur	iv
A. Ein Anhang	vi

1. Einleitung

Heutzutage ist die Menschheit darauf fokussiert, die komplette Welt zu digitalisieren. Dabei existiert ein Grundsatz, alles, was digitalisiert werden kann, soll digitalisiert werden. Um dies zu realisieren, ist es von Nöten, überall Hardware und Software zu verbinden. Sei es nun das Handy, mit welchem durch nur einem klick die Bankdaten angezeigt werden können, oder ein selbstfahrendes Auto, welches einen Anwender selbstständig zum Ziel fährt. Dies sind nur zwei Beispiele von 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“ definiert.

Embedded Systems können dabei grundsätzlich zwischen zwei Plattformen unterschieden werden:

- Deeply Embedded System
- Open Embedded System

Deeply Embedded Systems sind die wesentlichen Bausteine des Internet of Things [1]. Die Anwendung, die bei Deeply Embedded Systems implementiert wird, basiert auf speziell angepassten Echtzeitbetriebssystemen, den Programmiersprachen C oder C++ und ganz speziellen GUI¹-Frameworks² wie zum Beispiel TouchGFX.

Anders als bei den Deeply Embedded Systems, die sehr auf speziellen Technologien aufbauen, bieten Open Embedded Systems eine höhere Flexibilität in Sachen Technologien an. Dem Programmierer ist also die Möglichkeit gegeben, unterschiedliche

¹Graphical User Interface

²In der Softwareentwicklung ist ein Framework ein Entwicklungsrahmen, der dem Anwendungsprogrammierer zur Verfügung steht, um die grundlegende Architektur der Software zu bestimmen [2].

Technologien sowohl als auch unterschiedliche Programmiersprachen zu verwenden. Dort gilt bis dato die Kombination von C++ und Qt für GUI-lastige Systeme als „State of the Art“.

Die Konstellation zwischen C++ und Qt hat bislang auch funktioniert, jedoch kommt dieser Ansatz auch mit Problemen mit sich, denn die höheren Entwicklungszeiten für die Entwicklung von C++ Anwendungen sowie die geringe Verfügbarkeit von Experten auf dem Arbeitsmarkt sorgen für schlechtere Qualität und längere Produktionszeiten.

Um diesen Problemen zu entgegnen, soll in dieser Abschlussarbeit ein anderer Ansatz betrachtet werden. Und zwar könnten sowohl die Anwendungsschicht als auch die Persistenzschicht als .Net Core Anwendungen implementiert werden. Als GUI-Technologie soll dabei die neue Microsofttechnologie namens *Blazor* als Qt Ersatz zum Einsatz kommen. Somit kann erreicht werden, die komplette Codebasis mit .Net Core auszutauschen und eine Programmier-freundlichere Umgebung für Entwickler im Open Embedded Systems zu erschaffen.

1.1. Aufgabenstellung

Das Ziel dieser Arbeit ist die Entwicklung eines Blazor-basierten Frontend auf einem Raspberry Pi 4 um einen aussagekräftigen Vergleich zwischen den Technologien schaffen zu können und um eine mögliche verdrängung mittels Blazor zu demonstrieren. Dazu soll zunächst begutachtet werden, wie der momentane Stand der Technik für Open Embedded Systems ist, um anschließend das gleiche Verhalten mittels Blazor zu reproduzieren. Insbesondere sollen dabei verschiedene Aspekte, wie zum Beispiel das Verhalten zur Laufzeit, dieses Ansatzes überprüft werden.

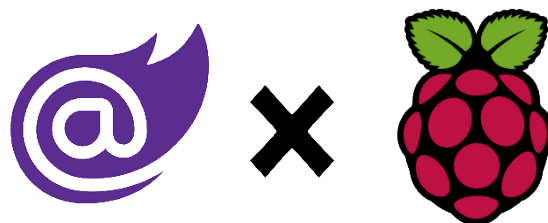


Abbildung 1.1: Blazor mit Raspberry Pi

1.2. Verwendet Hardware

Als Zielplattform für diese Thesis dient ein Raspberry Pi 4 B. Dieses wird unter anderem deswegen verwendet, da es im Labor Embedded Systems 2 der Hochschule Offenburg verwendet wird, aber auch, da es sehr gut im Open Embedded Systems bereich eingesetzt werden kann. Der Raspberry Pi 4 B verfügt dabei, unter anderem, über die folgenden technischen Spezifikationen: [3]

- 1,5 GHz ARM Cortex-A72 Quad-Core-CPU
- 1 GB, 2 GB oder 4 GB LPDDR4 SDRAM
- Gigabit LAN RJ45 (bis zu 1000 Mbit)
- Bluetooth 5.0
- 2x USB 2.0 / 2x USB 3.0
- 2x microHDMI (1x 4k @60fps oder 2x 4k @30fps)
- 5V/3A @ USB Typ-C
- 40 GPIO Pins
- Mikro SD-Karten slot

Und 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. Mit hilfe 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 SENSE HAT noch eine 8x8 LED-Matrix enthalten und ein Joystick mit 5 knöpfen, die angesteuert werden können.

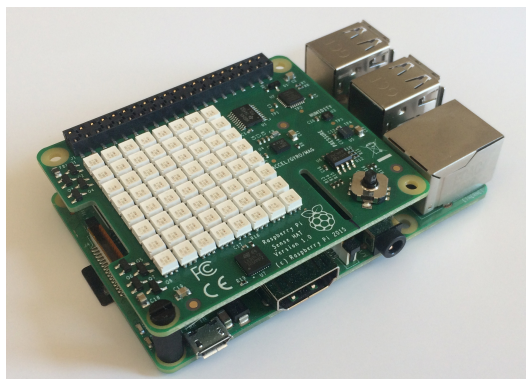


Abbildung 1.2: Verwendeter Raspberry Pi

1.3. Verwendete Software

Im Rahmen dieser Thesis 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 Viewer* verwendet. Dadurch ist die Möglichkeit gegeben, bequem und einfach den Raspberry Pi über einen Bildschirm zu bedienen.
- Für die Demonstration 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.
- Eine ausschlaggebende Technologie dieser Thesis wird durch das Framework *Blazor* von Microsoft abgebildet.
- Die Programmiersprachen dieser Thesis werden sich hauptsächlich aus C++ und C# beziehungsweise .Net zusammensetzen.
- Um den Zugriff auf die Funktionalitäten des *RPI SENSE HAT Shield* mittels .Net zu gewährleisten, wird die Iot Bibliothek von Microsoft verwendet.
- Die Implementierung der Blazor Anwendung wird dann letztendlich mittels der kostenlosen IDE *Visual Studio Code* realisiert.

Die oben vorgestellten Tools und Programmiersprachen wurden nicht explizit vorgegeben, sind dementsprechend auch selbst ausgesucht und sind zu Beginn dieser Thesis auch 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 momentane Entwicklung bei eingebetteten Systeme vonstattengeht, um anschließend eine beispielhafte Anwendung zu implementieren. Zudem soll auch das Framework dieser Thesis vorgestellt werden.

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. Dabei setzt sich ein eingebettetes System auch aus dem Zusammenspiel zwischen Hardware und Software zusammen. Solche eingebetteten Systeme haben meist kein ausgeprägtes Benutzerinterface und können weitergehend in die zwei unterklassen, Open und Deeply Embedded Systems unterteilt werden.

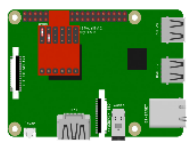
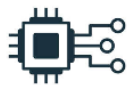
Open Embedded	Deeply Embedded
	
32/64 Bit Multicore Prozessor	8/16 Bit Singlecore Prozessor
Systemsoftware Applikation	Monitorprogramm Applikation
Für komplexe Aufgaben gedacht	Für einfache Aufgaben gedacht

Abbildung 2.1: Open vs Deeply Embedded Systeme [4][vgl.]

Neben der logischen Korrektheit die eingebettete Systeme an den Tag legen müssen, lassen sie sich durch eine Reihe unterschiedlicher Anforderungen und eigenschaften von den heutzutage üblichen Anwendungen abgrenzen. Unter anderem wird bei eingebettete Systeme ein sogenanntes *Instant on* gefordert, welches besagt, dass das Gerät unmittelbar nach dem Einschalten betriebsbereit sein muss [4][vgl.].

Weitere Anforderungen die bei Embedded Systemen zustande kommen, sind in der Folgenden Tabelle abgebildet:

Anforderung	Beschreibung
Funktionalität	Die Software muss schnell und korrekt sein
Preis	Die Hardware darf nicht zu kostenspielig 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, mehr als 30 Jahre zu laufen ohne groß zu verschlechtern

Tabelle 2.1.: Anforderungen an eingebettete Systeme [4]

2.1.1. Hardware

Die einzelnen Komponenten, die in einem eingebetteten System verbaut worden sind, entscheiden über die vorhandene Leistung, den Stromverbrauch und die Robustheit, die dieses System ausmachen. Die kern Komponente eines eingebetteten Systems wird durch einen Prozessor repräsentiert und werden häufig als *System on Chip* eingebaut. Dabei ist die Tendenz zu den ARM-Core Modellen steigend. Neben dem Prozessor befinden sich typischerweise auf den eingebetteten Systemen noch weitere Komponenten, wie zum Beispiel dem Hauptspeicher, dem persistenten Speicher und diversen Schnittstellen, um weitere peripherie Geräte anzuschließen [4][vgl.].

Damit weitere peripherie Geräte 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 periphere Geräte zu verbinden. Nicht selten kommt es vor, dass in einem eingebetteten System darüber hinaus noch eine galvanische Entkopplung¹ eingebaut wird, damit die Hardware vor Störungen von außen, wie Beispiel Motoren, geschützt ist [4][vgl.].

2.1.2. Software

Genauso, wie sich eingebettete Systeme in zwei Bereiche unterscheiden lassen können, kann die Software eines eingebetteten Systems in Systemsoftware und funktionsbestimmende Anwendungssoftware unterteilt werden. Da *Deeply Embedded Systems* meist nur für kleine und einfache Aufgaben ausgelegt sind, benötigen diese meist nur sehr schwache Software. Diese funktionsbestimmende Anwendungssoftware ist dann meist ein spezielles Echtzeitbetriebssystem, welche auf die Aufgabe und der Hardware angepasst ist.

Ganz anders sieht es im *Open Embedded Systems* Bereich aus, welche seine Software als Kennzeichen hat. Da diese für sehr komplexe Aufgaben zum Einsatz kommen, kommt es nicht selten vor, dass auch eine GUI für ein solches System vonnöten ist. Deswegen basiert ein *Open Embedded Systems* auf einer Systemsoftware, die dem Programmierer mehr Möglichkeiten beim Entwickeln gibt. Unter anderem ist somit auch die Möglichkeit gegeben, die Programmiersprache und das GUI-Framework, nach Belieben selbst auszusuchen und nicht auf spezielle Software angewiesen zu sein [4][vgl.].

2.2. Qt

In der vorherigen Sektion wurde ein allgemeines Bild dargestellt, was ein eingebettetes System ist und in welchen Varianten diese auftauchen. Weitergehend soll nun, in dieser Sektion, vorgestellt werden, mit dem üblicherweise im *Open Embedded Systems* Bereich programmiert wird.

Ein großer Anteil eines *Open Embedded Systems* wird heutzutage durch die GUI repräsentiert. Die GUI sollte intuitive und zuverlässig sein. Zudem ist es auch noch

¹Unter der galvanischen Entkopplung versteht man das Vermeiden der elektrischen Leitung zwischen zwei Stromkreisen, zwischen denen Leistung oder Signale ausgetauscht werden sollen [5]

enorm wichtig, dass die GUI wenige Ressourcen verbraucht, schnell reagiert und einfach einzubinden ist. Um diese Anforderung zu erreichen, wurde bis dato *Qt* in Kombination mit der Programmiersprache *C++* verwendet [6][vgl.].

2.2.1. Was ist Qt?

Qt ist ein Framework zum Erzeugen von GUI's auf mehreren Betriebssystemen. Es wurde 1990 von *Haarvard Nord* und *Eirik Chambe-Eng* unter dem Vorwand entwickelt, benutzer freundliche GUI's mithilfe von der Programmiersprache *C++* zu entwickeln. Der Name *Qt* entstand dabei, da *Haarvard* den Buchstaben *Q* in *Emacs*² als sehr schön empfand. Zudem entstand das *t* in *Qt* als abkürzung für das englische Wort *Toolkit* [8][vgl.].

Die Intension hinter Qt war es damals nicht nur ein Framework zu erschaffen, welches benutzer freundliche GUI's kreiert, sondern dies auch, mit nur einer Code-Basis über alle Betriebssysteme hinweg. Die schwierigkeit bestand also darin, das selbe Aussehen, die selbe benutzer Erfahrung und die gleiche Funktionalität über die verschiedenen Betriebssysteme zu schaffen [9][vgl.].

Qt ist in *C++* entwickelt worden und verwendet zusätzlich noch einen Compiler³ welcher *moc*⁴ genannt wird, womit *C++* um weitere Elemente erweitert wird. Der daraus compilierte Code folgt dem *C++*-Standard und ist somit mit jeden anderen *C++* Compiler kompatibel. Obwohl Qt ursprünglich dafür gedacht war, rein auf *C++* zu basieren, wurden im Laufe der Zeit mehrere Erweiterungen für Qt von der Community entwickelt, um Qt mit mehreren Programmiersprachen benutzen zu können, wie zum Beispiel Python oder Java.



Abbildung 2.2: Qt Logo

²Emacs ist ein leistungsfähiger Texteditor und im Unterschied zu den meisten Editoren eine komplette Arbeitsumgebung [7]

³Ein Compiler, im einfachen sinne, ist ein Übersetzer, der von einer Programmiersprache in Computersprache (0 und 1) Übersetzt [10][vgl.].

⁴meta object compiler

2.2.2. Programmierbeispiel

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

```
1 #include "mainwindow.h"
2
3 int main(int argc, char *argv[])
4 {
5     // Set the Application
6     QApplication app(argc, argv);
7     QWidget window;
8     // Set fixed Width of the Window
9     window.setFixedWidth(300);
10
11    // Set the Title of the Window
12    window.setWindowTitle("Meine erste Qt App");
13
14    // Create a Label and align it to Center
15    QLabel *lblHello = new QLabel("Hello World!");
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

Das Programm welches erzeugt wird, wird folgend dargestellt:

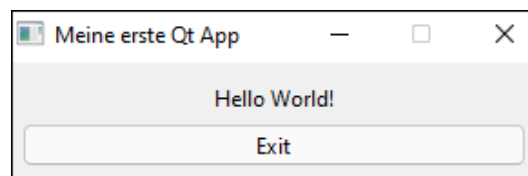


Abbildung 2.3: Qt Hello World App

2.2.3. Widgets

Um eine GUI gestalten zu können braucht es Komponenten, die auf der GUI angezeigt werden können. Qt verwendet dafür sogenannte *Widgets*. Widgets sind als grafische Komponenten, die dafür genutzt werden können, um die Benutzeroberfläche nach Belieben 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 vorgestellt werden: Wie zu sehen ist, ist ganz oben in der Klassenhierarchie die *QObject* Klasse. Diese

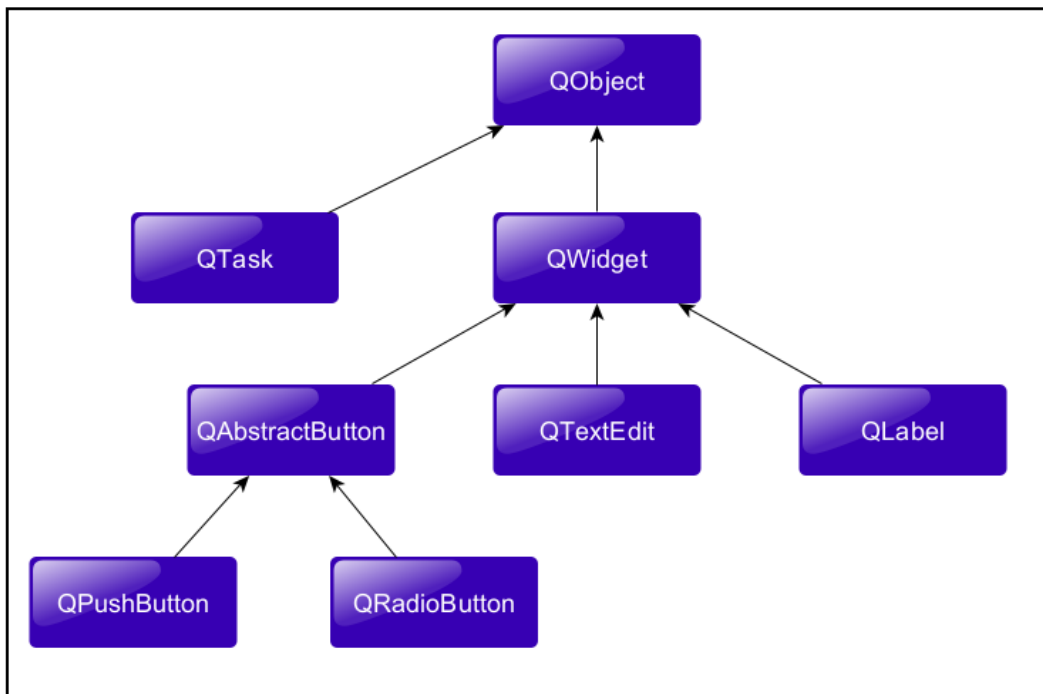


Abbildung 2.4: Qt Widgets Klassenhierarchie [9][vgl.]

enthält unter anderem den Signal und Slot mechanismus, der Später noch genau erklärt wird. Weitergehend, werden Widgets die gemeinsame Funktionalitäten aufweisen zusammen gruppiert. Diese Verhalten ist bei *QPushButton* und *QRadioButton* erkennbar, denn beide Widgets sind Buttons, die sich Teilweise die gleichen Eigenschaften und Funktionen teilen [9][vgl.].

2.2.4. Signal und Slot Konzept

Um eine Benutzeroberfläche wirklich interaktiv zu gestalten, ist es vonnöten, auf bestimmte Ereignisse zu reagieren. 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-Aktions Konzept implementiert werden kann.

Qt benutzt um diese Ereignis-Aktion Muster zu bewerkstelligen, das selbst entwickelte Signal und Slot Konzept. Ein *Signal* ist im einfachen sinne eine Nachricht, die Versendet wird, um zu signalisieren, dass der momentane Status eines Objektes sich geändert hat. Dahingegen ist ein *Slot* eine spezielle Funktion von einem Objekt, welche immer dann aufgerufen wird, wenn ein bestimmtes *Signal* gesendet wird [9][vgl.].

Damit jeder *Slot* auch weiß, auf welches *Signal* reagiert werden soll, müssen diese zusammen 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. Dabei wurde sich mit dem Button *btnExit* verbunden und ein Signal gesendet, wenn der Button betätigt wird. Weitergehend wurde sich noch mit *app* verbunden, auf welchem dann, beim Betätigen von *btnExit*, die Funktion *quit* aufgerufen wird. Sobald also das Signal gesendet wird, schließt sich die Application.

Einer der größten Vorteile dieser Methode im gegensatz zu anderen Methoden, ist es, dass dadurch N zu M Beziehungen abgebildet werden können. Das bedeutet also, dass sich ein Signal mit beliebig viele Slots verbinden kann und dass sich ein Slot auf mehrere Signale verbinden kann [11][vgl.].

2.2.5. Raspberry Pi und Qt

test

3. Blazor

Nachdem nun, in der vorherigen Sektion, das C++ Framework Qt vorgestellt wurde, soll in dieser Sektion 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. Der Name *Blazor* entstand aus der Kombination der zwei Wörter *Browser* und *Razor*. Browser zum einen, da das Ziel war, C# in den Browser zu bekommen und Razor zum anderen, da Blazor gebrauch von der Razor syntax macht [12][vgl.]. Es wurde 2019 erstmalig von Microsoft veröffentlicht, mit der Intension Webseiten oder auch SPA's¹ mithilfe von C# zu entwickeln. Dabei existieren 2 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 native im Browser läuft, dazu aber im späteren verlauf dieser Thesis mehr [13][vgl.].

Die Idee, die hinter dem Framework von Microsoft steckt, ist es nicht nur C# in den Browser zu bekommen, um somit Javascript zu verdrängen, sondern auch mit nur einer Codebasis sowohl im Frontend, als auch im Backend zu entwickeln. Somit wurde geschaffen, dass langjährige C# Entwickler mit ihrem vorhandenen Wissen als Full-Stack entwickler, eingesetzt werden können.

¹Single Page Applications

3.2. Architekturen

Da Blazor in zwei Varianten existiert, existieren dementsprechend auch zwei Architekturen für dieses Framework. Bevor die zwei Architekturen jedoch im Detail erklärt werden, sollte zuerst ein grundlegendes Wissen darüber veranschaulicht werden, wie bis jetzt andere SPA's, wie zum Beispiel Angular oder React sowas gehandhabt haben. Die heutigen SPA's basieren auf der Client-Server Architektur, das bedeutet der Client stellt eine Anfrage an den Server und erhält dann, sollte alles stimmen, die passende Antwort. Die Kommunikation der beiden Teilnehmer geschieht dann in den meisten Fällen über eine REST Api. Heutzutage gibt es noch andere Alternativen zu REST, wie zum Beispiel gRPC, jedoch ist bis heute der Standard für die Kommunikation noch REST. In dem folgenden Schaubild ist eine solche Architektur zu sehen:

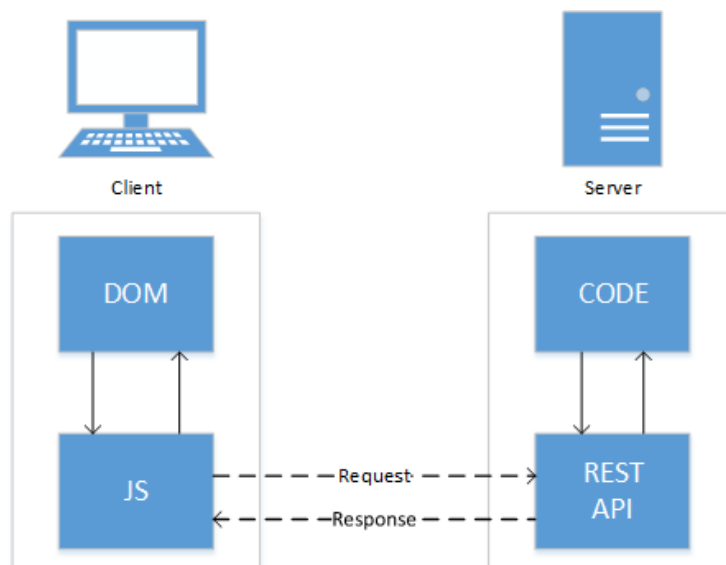


Abbildung 3.1: Client-Server Architektur mit Javascript

Wie zu sehen ist, kommuniziert der Client mithilfe von Javascript mit dem Server. Javascript holt sich also die Daten, welche der Client braucht und gibt diese dem DOM zum Verarbeiten weiter.

3.2.1. Blazor WebAssembly

Bei Blazor WebAssembly ist es so, dass sich die Architektur von den anderen SPA's wie Angular nicht groß verändert. Tatsächlich verändert sich hierbei nur die Pro-

grammiersprache, die auf dem Client läuft. Es handelt sich dabei um die Programmiersprache C#, die dann auf dem Client ausgeführt wird, wie im folgenden zu sehen ist.

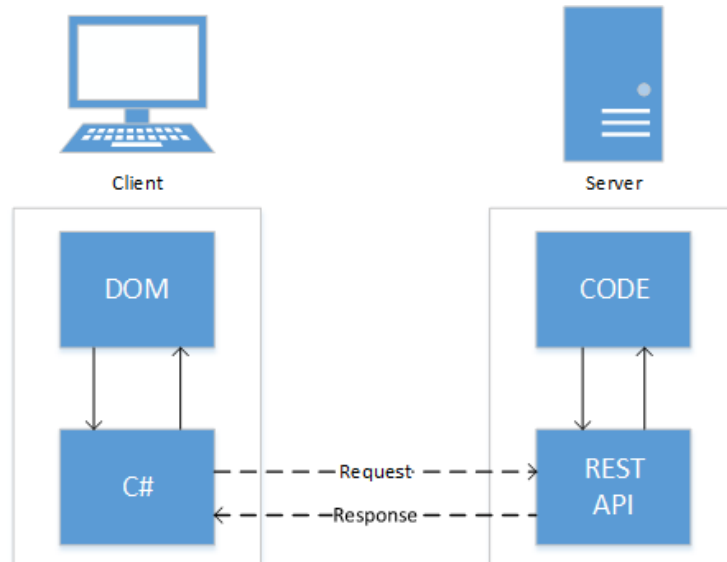


Abbildung 3.2: Blazor WebAssembly Architektur

Das ganze Konzept C# überhaupt auf dem Client laufen lassen zu können, funktioniert durch WebAssembly. WebAssembly wandelt Programmcode in nativen Bytecode um, der dann in einer Sandbox im Browser ausgeführt werden kann. Dabei wird von der Sandbox aus die DOM mit Hilfe von Javascript kontinuierlich manipuliert. Javascript verschwindet in dem Sinne also nicht komplett sondern wird lediglich ergänzt. Da für dieses Konzept als WebAssembly vonnöten ist, kann Blazor WebAssembly nicht auf Browsern funktionieren, die WebAssembly nicht unterstützen [12][vgl.].

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

- + Sehr Skalierbar
- + Sehr Performant
- + Es kann komplett eigenständig auf dem Client laufen und ist nicht unbedingt auf dem Server angewiesen
- Große Anwendungsdatei, die komplett auf dem Client geladen werden muss

- Lange ladezeit beim ersten Aufruf
- Kompletter Code ist auf dem Client zu sehen

3.2.2. Blazor Server

Anders als es bei Blazor WebAssembly der Fall ist, wird bei Blazor Server nicht C# in den Browser geladen, sondern das ganze Konzept spielt sich auf dem Server ab. Deswegen wird auch die Web-Anwendung als SPA auf dem Server gerendert. Zum Client werden nur JavaScript und Markup gesendet und die Daten und Benutzereingaben laufend mittels SignalR zwischen Client und Server ausgetauscht. Dementsprechend ist es auch vonnöten, dass immer zwischen dem Client und dem Server eine offene Verbindung vorhanden ist [12][vgl.].

Dadurch das die komplette Seite auf dem Server gerendert wird, lädt die Seite auf dem Client sehr schnell, muss nur eine kleine Javascript Datei auf den Client laden und kann auf sehr Leistungsschwachen Clients verwendet werden. Zudem kommt auch noch, anders als bei Blazor WebAssembly, dass sich jeder Browser verwenden lässt unabhängig davon ob dieser WebAssembly unterstützt oder nicht.

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 Daten kommuniziert dann mit dem DOM und baut die Verbindung zum Server auf. Sobald die Verbindung aufgebaut ist, werden kontinuierlich Nachrichten zwischen Client und Server ausgetauscht, wie im folgenden zu sehen ist:

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

- + Kurze Ladezeiten
- + Es ist komplett Browserunabhängig
- + Client hat keinen Zugriff auf den SourceCode
- Nicht Skalierbar, da alle Benutzer auf einem Server zugreifen
- Lange Netzwerklatenz führt zu Verzögerungen in der Benutzeroberfläche
- Es muss immer eine Verbindung bestehen

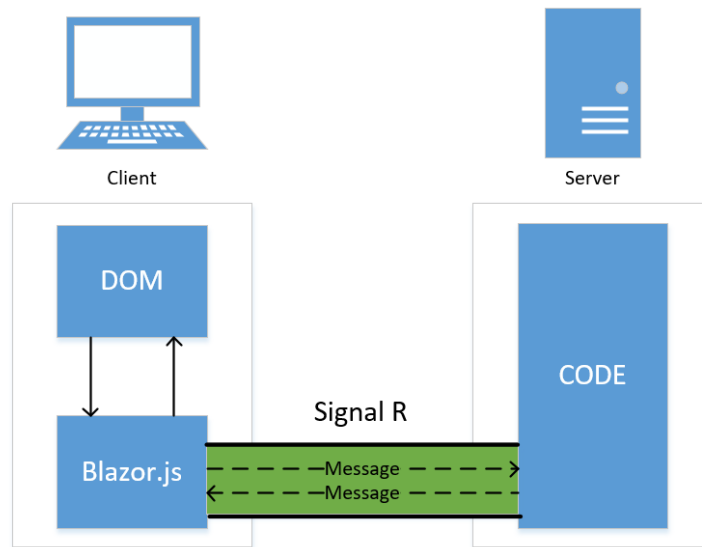


Abbildung 3.3: Blazor Server Architektur

3.3. Komponenten

Anders als es bei Qt der Fall ist, basiert Blazor wie andere Web-Frameworks auf *Html* und *Css*. Das bedeutet, dass der Entwickler auf jedes Html element zugreifen kann, das existiert. Weitergehend hat der Entwickler noch die möglichkeit auf zusätzliche Infragistics wie zum Beispiel *MadBlazor* oder *Ignite UI* zurückzugreifen.

Außerdem bietet Blazor zusätzlich noch die Möglichkeit eigene Komponenten zu erstellen. Eine Komponente, im Sinne von Web-Frameworks, kann sich so wie eine Methode in einer Programmiersprache vorgestellt werden, das heißt, es kann eine Html und Css Sequenz ausgelagert werden, um diese an mehrere Stellen zu verwenden. Hinzukommt kann eine Komponente, in Blazor, in zwei Varianten vorkommen. Einmal als eine *Page-Komponente* und einer *Non-Page-Komponente*. Der Unterschied zwischen den beiden ist, dass man eine Page-Komponente durch eine URL adressieren kann, das heißt, dass diese Komponente als eigenständige Seite einer Webseite angesehen werden kann und eine Non-Page-Komponenten, wirklich nur die Sequenz von Html und Css darstellt.

```
1 @page "/counter"
2
3 <PageTitle>Counter</PageTitle>
4
5 <h1>Counter</h1>
6
7 <p role="status">Current count: @currentCount</p>
8
```



```
9 <button class="btn btn-primary" @onclick="IncrementCount">Click me</button>
10
11 @code {
12     private int currentCount = 0;
13
14     private void IncrementCount()
15     {
16         currentCount++;
17     }
18 }
```

Listing 3.1: Page-Komponente Beispiel

Listing 3.1 zeigt eine Page-Komponente, die sowohl einen Text und einen Button anzeigt und den momentane wert von *currentCount* um eins erhöht wenn der Button betätigt wird. Bei der Komponente handelt es sich um eine Page-Komponenten, da dies mit dem *@page* in der ersten Zeile ausgezeichnet ist.

Einer Komponente, sei es nun eine Page-Komponente oder Non-Page-Komponente, können auch einen oder mehrere Parameter von der Oberkomponente mitgegeben werden. Dies hat den Vorteil, dass Komponenten mehr an flexibilität gewinnen und somit besser wiederverwendbar sind, wie im folgenden Beispiel zu sehen ist:

```
1 <h2>@Title</h2>
2
3 @code {
4     [Parameter]
5     public string Title { get; set; }
6 }
```

Listing 3.2: Kind Komponente

```
1 @page "/parent"
2
3 <h1>Parent</h1>
4
5 <Child Title="Child-Title"/>
6
7 @code {
8
9 }
```

Listing 3.3: Eltern Komponente

Wie in Listing 3.2 zu sehen ist, hat diese Komponente einen Parameter, die sie dann, wie hier im Beispiel, als *Title* in einem *<h2>-Tag* darstellt. Listing 3.3 kann dann gebrauch von der Kind Komponente machen und ihr einen *Title* mitgeben. Der *Html Tag* der Kind Komponente, wird durch den Dateinamen der Komponente

erzeugt, das bedeutet, da die Komponente *Child.razor* im Projekt heißt, wird diese dann auch mittels dem *<Child>-Tag* verwendet.

3.4. Javascript Interoperation

Test

3.5. Blazor MAUI

Test

4. Raspberry Pi mit .Net Core und Blazor

Test

4.1. Entwicklungsumgebung

Test

4.2. Installation

Test

4.3. Programmieren mit .Net Core

Test

4.4. Blazor Demo Anwendung

test

5. Analyse

Test

6. Fazit

Test

7. Ausblick

Test

Abkürzungsverzeichnis

GUI	Graphical User Interface	1
moc	meta object compiler	8
SPA 's	Single Page Applications	12

Tabellenverzeichnis

2.1. Anforderungen an eingebettete Systeme	6
A.1. Tabellenunterschrift	vi

Abbildungsverzeichnis

1.1. Blazor mit Raspberry Pi	2
1.2. Raspberry Pi 4 B	3
2.1. Open vs Deeply Embedded Systeme	5
2.2. Qt Logo	8
2.3. Qt Hello World App	9
2.4. Qt Widgets Klassenhierarchie	10
3.1. Client-Server Architektur mit Javascript	13
3.2. Blazor WebAssembly Architektur	14
3.3. Blazor Server Architektur	16
A.1. Beschreibung für Verzeichnis2	vii

Listings

2.1. Qt Hello World Sourcecode	9
2.2. Signal und Slot beispiel	11
3.1. Page-Komponente Beispiel	16
3.2. Kind Komponente	17
3.3. Eltern Komponente	17

Literatur

- [1] Hochschule niederrhein, *CAS Embedded Systems Professional - Hochschule Niederrhein*, 29.10.2021. Adresse:
<https://www.hs-niederrhein.de/weiterbildung/sichere-software/cas-embedded-systems-professional/>.
- [2] Business Systemhaus AG, *Was ist ein Framework? Definition & Erklärung - BSH AG*, 16.11.2021. Adresse:
<https://www.bsh-ag.de/it-wissensdatenbank/framework/>.
- [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] 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.
- [5] Breimer-Roth Transformatoren, *Galvanische Trennung: Wissenswertes zur galvanischen Trennung bei Breimer-Roth*, 10.05.2021. Adresse:
<https://bre-trafo.de/wissen/galvanische-trennung/>.
- [6] 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.
- [7] D. Cameron, *GNU Emacs: Kurz & gut*, 1. Aufl., 2., korr. Nachdr., dt. Ausg., Ser. O'Reillys Taschenbibliothek. Beijing und O'Reilly, 2000, ISBN: 3897212110.
- [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] t2informatik. Wir entwickeln Software., „Was ist ein Compiler? - Wissen kompakt - t2informatik“, *t2informatik GmbH*, 10.07.2018. Adresse:
<https://t2informatik.de/wissen-kompakt/compiler/>.
- [11] 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.
- [12] bbv, *Hier kommt Blazor*, 4.12.2021. Adresse: <https://www.bbv.ch/blazor/>.
- [13] 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>.

A. Ein Anhang

Referenz zu Tabelle A.1.

Bezeichnung	Typ	Beschreibung
load.load1	float	The load average over 1 minute.
load.load5	float	The load average over 5 minutes.
load.load15	float	The load average over 15 minutes.
cpu.user	int	The amount of CPU time spent in user space.
cpu.user_p	float	The percentage of CPU time spent in user space. On multi-core systems, you can have percentages that are greater than 100%. For example, if 3 cores are at 60% use, then the cpu.user_p will be 180%.
cpu.system	int	The amount of CPU time spent in kernel space.
cpu.system_p	float	The percentage of CPU time spent in kernel space.
mem.total	int	Total memory.
mem.used	int	Used memory.
mem.free	int	Available memory.
mem.used_p	float	The percentage of used memory.

Tabelle A.1.: Tabellenunterschrift

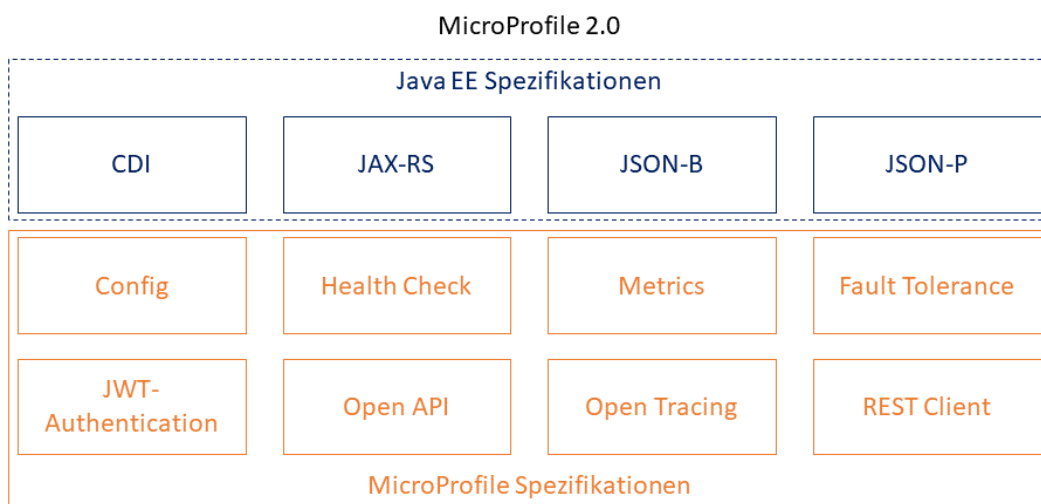


Abbildung A.1: Bildunterschrift2