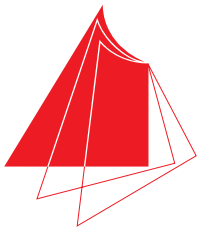


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Entwurf einer Architektur und eines Proof of Concept für ein echtzeitfähiges SCADA System mit Webfrontend

Bachelor-Thesis

zur Erlangung des akademischen Grades
Bachelor of Engineering

vorgelegt von

Florian Weber (44907)

12.11.2019

Erstprüfer: Prof. Dr.-Ing. Philipp Nenninger
Zweitprüfer: Prof. Dr. Stefan Ritter

Inhaltsverzeichnis

Abkürzungsverzeichnis

DBMS Datenbank Management System

crud Create, read, update and delete

TCP Transmission Control Protocol

CSMA/CD Carrier Sense Multiple Access/Collision Detection

PoC Proof of Concept

SCADA Supervisory Control And Data Acquisition

HTTPS Hypertext Transfer Protocol Secure

HTTP Hypertext Transfer Protocol

WSS WebSocket Secure

OPC UA Open Platform Communications Unified Architecture

HTML Hypertext Markup Language

JS JavaScript

CSS Cascading Style Sheets

SQL Structured Query Language

GUI Graphical User Interface

PLC Programmable Logic Controller

ERD Entity Relationship Diagram

CA Certification Authority

URL Uniform Resource Locator

IP Internet Protocol

JSON JavaScript Object Notation

STL Standart Template Library

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einführung

1.1 Motivation

Durch mein Studium der Elektrotechnik mit Vertiefung in die Automatisierungstechnik konnte ich Eindrücke in die Vorgehensweise und Möglichkeiten der Graphical User Interface (GUI) Programmierung für Industrieanlagen gewinnen. Dabei fiel mir auf, dass der aktuelle Stand der Technik in der Automatisierungstechnik noch stark vom Stand in anderen softwaregeprägten Bereichen abweicht. So wird in der Automatisierungstechnik noch immer auf statisch geschriebene Benutzeroberflächen gesetzt, die kompilliert werden müssen und damit viele Einschränkungen mit sich bringen. Es ist zum Beispiel nicht möglich ein Steuerelement zur Laufzeit in Abhängigkeit vorhandener Entitäten zu instanzieren. Meist schafft man sich Abhilfe, indem man ein Element entweder ein- oder ausblendet. Ein weiteres Problem vorhandener Lösungen ist, dass diese meist plattformgebungen sind und nur lokal im Netz, mit entsprechender Software des Herstellers, lauffähig sind. In der heutigen Informatik wird immer mehr auf grafische Benutzeroberflächen gesetzt, welche als Webapplikation in einem beliebigen Browser verwendbar sind.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist der Entwurf einer Architektur für ein echtzeitfähiges Supervisory Control And Data Acquisition (SCADA) System mit Webfrontend. Durch ein Proof of Concept (PoC), wird herausgefunden ob die Architektur auch in die Praxis umsetzbar ist. Hierbei wird die Applikation streng in Frontend und Backend getrennt. Das Frontend wird als Webapplikation im PoC implementiert. Dabei werden folgende Anforderungen an die Architektur gestellt:

- Die Datenrate des Frontends soll bei vertretbarem Aufwand so klein wie möglich sein. Dies ermöglicht die Nutzung des Systems in einem Umfeld mit geringer verfügbarer Bandbreite zur Steuerungsebene.
- Die Architektur soll Steuerelemente unterstützen, die eine Eingabe durch den Nutzer zulassen, sowie Steuerelemente die eine Darstellung eines Prozesswerts ermöglichen.

- Die Webapplikation selbst soll so modular sein, dass man zur Laufzeit Steuerelemente hinzufügen und entfernen kann, ohne dass das System offline geht.
- Die Prozessdaten sollen nicht, wie aktuell bei vielen Webapplikationen üblich, durch Polling synchronisiert werden, sondern die Weboberfläche soll auf Datenänderungen des Prozesses asynchron in Echtzeit (bei statischem Routing im Netzwerk) reagieren. Dasselbe gilt für die Eingaben des Nutzers.
- Die Architektur soll eine herstellerunabhängige Schnittstelle zur Integration in ein vorhandenes System bereitstellen.
- Die Weboberfläche soll eine feste Auflösung haben und muss nicht auf Änderungen des Viewports reagieren. Ausnahmen bilden hierbei Darstellungen die dies, durch ihre einfache Gestalt, erlauben.

Der Beweis der Realisierbarkeit soll durch eine Beispielimplementierung (PoC) erbracht werden. Dabei wird je ein Eingabeelement (z.B. Button), ein Ausgabeelement (z.B. Label), sowie ein Ein-/Ausgabeelement (z.B. ein Textfeld) implementiert.

1.3 Gliederung

2 Theoretische Grundlagen

2.1 Security

2.1.1 Authentifizierung

2.1.2 Verschlüsselung

2.2 Websocket

Websockets bieten die Möglichkeit Nachrichten zwischen einem Server und einer Webanwendung asynchron auszutauschen. Jede Websocket Verbindung startet zu Beginn als Hypertext Transfer Protocol (HTTP) Verbindung. Der Client sendet einen HTTP Request vom Typ *GET*, mit den Attributen *Connection: Upgrade* und *Upgrade: websocket* im Header, an den HTTP Server. Unterstützt der HTTP Server das Websocket Protokoll, bestätigt er die Anfrage mit einem *HTTP/1.1 101 Switching Protocols* Response. Anschließend bauen die Teilnehmer eine Transmission Control Protocol (TCP) Verbindung auf, die so lange bestehen bleibt, bis einer der Teilnehmer die Verbindung aktiv beendet oder Ein Teilnehmer nicht mehr ereichbar ist. Die ausgetauschten Daten können Strings sein oder binäre Daten. Beides wird vom Websocket Protokoll unterstützt. Das Protokoll definiert allerdings keine Semantik der übertragenen Daten, weshalb immer ein Subprotokoll nötig ist, um damit produktiv zu arbeiten. Wie auch eine HTTP Verbindung, kann eine Websocket Verbindung verschlüsselt werden. Dies erkennt man am Protokollpräfix des Uniform Resource Locator (URL). Bei einer unverschlüsselten Verbindung lautet der Präfix „*ws://*“ und bei einer verschlüsselten Verbindung „*wss://*“.

2.3 Datenbanken

Der Zweck eines Datenbanksystems ist es Daten persistent zu speichern und zur Verfügung zu stellen. Ein Datenbanksystem setzt sich aus einer Datenbasis, sowie einem Datenbank Management System (DBMS) zusammen. Dabei stellt die Datenbasis den Speicher der Datenbank dar und das DBMS das Programm durch das der Zugriff auf die Datenbasis ermöglicht wird. DDas DBMS ist notwendig, um konkurrierende Zugriffe von mehreren Nutzer auf die selbe Datenbasis zu verwalten [3]. Meistens bietet das DBMS als Interface eine Netzwerkschnittstelle

an (Server-Client Architektur), es gibt allerdings auch Datenbanken deren DBMS Teil der Applikation werden (z.B. SQLite).

2.3.1 Relationale Datenbank

Im Falle einer relationalen Datenbank werden Daten in Form von Relationen abgespeichert. Tabellen sind eine Darstellung von Relationen.
Jede Relation hat Attribute mit einem in der Relation einzigartigem Attributnamen. Die Domänene eines Attributs ist die Menge aller Werte die ein Attribut annehmen kann. Ist der Wert eines Attributs unbekannt oder noch nicht bestimmt, ist der Wert des Attributs „*NULL*“ [4]. Werden Daten nun in einer Datenbank gespeichert, wird ein Wertetupel in eine Relation eingefügt. Die Relationen sind nur das strukturelle theoretische Konzept, das sich hinter einer relationalen Datenbank verbirgt. Anfangs dieses Abschnitts wurde gesagt, dass Relationen in Form einer Tabelle darstellbar sind. Wenn Relationen nun zu ihrer Definition Wertetabellen zu Grunde liegen, ist es möglich zur Vereinfachung statt von Relationen von Tabellen zu sprechen. Das DBMS stellt eine Schnittstelle bereit um dem Nutzer die Daten zugänglich zu machen (Create, read, update and delete (crud) Operationen). Die Schnittstelle unterstützt meist die Sprache Structured Query Language (SQL). Diese Sprache ist in Abschnitt 2.3.2 genauer beschrieben. Relationen können nicht nur als Tabelle ausgegeben werden, es ist auch möglich sie zu verknüpfen. Wenn keine Regel angegeben wird wie zwei Relationen Verknüpft werden sollen, wird das informatische Kreuzprodukt gebildet. Das bedeutet, dass jedes Wertetuple aus Relation A, mit jedem Wertetupel aus Relation B verknüft wird. Dadurch entsteht eine neue Relation. In einer Datenbank existieren immer Schlüssel (Keys) innerhalb einer Tabelle. Dabei gibt es drei Typen:

- Primary-Key (Primärschlüssel)
- Unique-Key
- Foreign-Key (Fremdschlüssel)

Der Primary-Key muss als einziger Schlüssel zwingend vorhanden sein. Er bestimmt nach was das DBMS die Datensätze ablegt. Deshalb muss der Wert des Attributes das den Primary-Key bildet, innerhalb einer Tabelle einzigartig sein. Das heist es darf maximal ein Wertetupel innerhalb einer Relation existieren, mit dem selben Wert des Primärschlüssels. Der Primärschlüssel darf auch nicht „*NULL*“ sein. Ein Unique-Key stellt ein zusätzlichen Schlüssel dar, über die jeder Eintrag in einer Tabelle eindeutig identifizierbar ist. Wie auch der Primärschlüssel, darf dieser Schlüssel nicht „*NULL*“ sein. Er ist in jeder Tabelle nur optional vorhanden und kann auch mehrfach vorkommen. Der Foreign-Key (Fremdschlüssel) ist ein Schlüssel, der nur in einem relationalen DBMS unterstützt wird. Er ermöglicht es, einen Verweises auf eine andere Tabelle zu

definieren. Zur Veranschaulichung sei folgendes Beispiel einer relationalen Datenstruktur, auf einem SQL Server, gegeben. Wie in Abbildung 2.1 zu sehen,

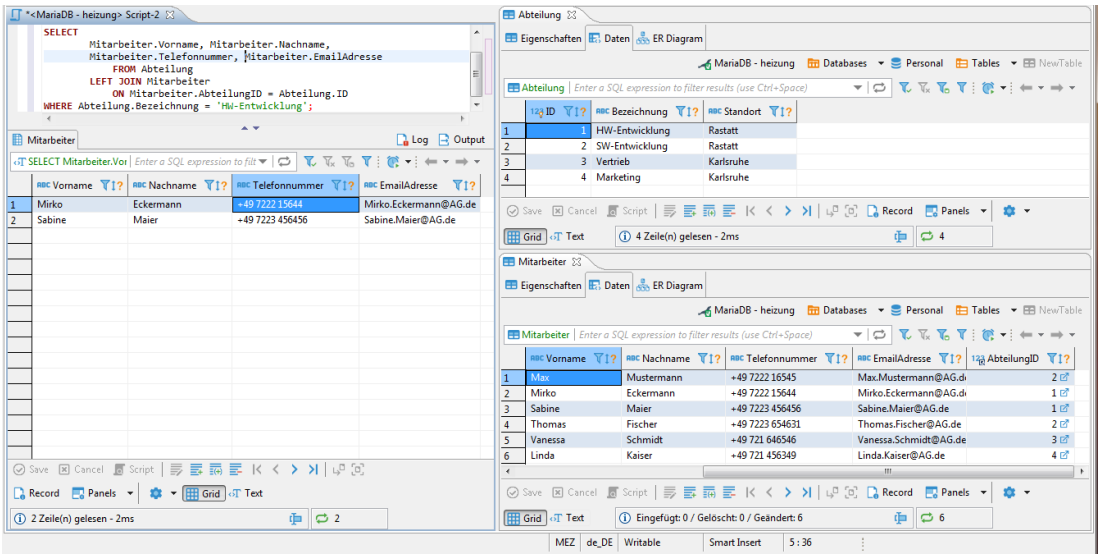


Abbildung 2.1: Beispiel Datenstruktur

besteht die Datenbank „*Personal*“ aus zwei Tabellen. Die erste Tabelle hat die Bezeichnung („*Abteilung*“) mit den Spalten „*ID*“, „*Bezeichnung*“ sowie Standort. Die Spalte „*ID*“ ist als Primary-Key deklariert. Die zweite Tabelle („*Mitarbeiter*“) enthält die Spalten „*Personalnummer*“, „*Vorname*“, „*Nachname*“, „*Telefonnummer*“, „*EmailAdresse*“ und „*AbteilungID*“. In dieser Tabelle ist die Personalnummer der Primärschlüssel. Um eine Verknüpfung der Spalte „*AbteilungID*“ der Tabelle „*Mitarbeiter*“ zu der Tabelle „*Abteilung*“ herzustellen, wird ein Fremschlüssel definiert, der von der Spalte „*AbteilungID*“ der Tabelle „*Mitarbeiter*“ auf die Tabelle „*Abteilung*“, unter Benutzung deren Primärschlüssels, verweist. Die Definition eines Fremdschlüssels ist nicht zwingend notwendig, um die Verknüfung der zwei Relationen nach dieser Semantik zu ermöglichen. Es ermöglicht aber die selbstständige Erhaltung der referenziellen Integrität durch das DBMS. Dabei hat man bei der Erstellung des Fremdschlüssels die Wahl, was beim Löschen oder beim Ändern eines referenzierten Datensatzes geschehen soll. Der gängigste Umgang damit ist, dass man entweder das Löschen verbietet (*on delete restrict*), oder das beim Löschen des referenzierten Datensatzes auch der referenzierenden Datensatz gelöscht wird (*on delete cascade*). Bei beiden Verfahren bleibt die referenzielle Integrität erhalten. Das heißt es gibt nach dem Löschen keine Referenzen, die nicht aufgelöst werden können.

2.3.2 SQL

SQL ist eine Abkürzung für Structured Query Language. Dabei handelt es sich um eine Sprache, welche das Erzeugen und Verwalten von Datenstrukturen ermöglicht. Außerdem ermöglicht sie das Abfragen, Einfügen, Verändern, sowie Verknüpfen von Datensätzen. Dabei unterscheidet sich SQL sehr von anderen Programmiersprachen. So besteht der Ansatz bei SQL eher darin zu definieren, was man als Ergebnis möchte, sich aber um die konkrete Implementierung der Operation keinerlei Gedanken machen braucht. Um dies zu demonstrieren ist das folgende Beispiel, auf Basis der Beispieldatenbank in Abbildung 2.1, gegeben. Nun möchte man alle Telefonnummern, Namen und E-Mail Adressen einer Abteilung haben, welche den Namen *HW-Entwicklung* trägt. Dazu ist es notwendig die Datensätze der Mitarbeitertabelle mit den Datensätzen der Abteilungstabelle, entsprechend des Fremsschlüssels in der Mitarbeiter Tabelle, zu kombinieren und alle Datensätze der so entstandenen Tabelle auszugeben, welche die Abteilungsbezeichnung *HW-Entwicklung* tragen. Die Query für diese Operation ist in Abbildung 2.2 abgebildet. Die Antwort des Servers ist in Abbildung 2.1 unten

```
SELECT
  Mitarbeiter.Vorname, Mitarbeiter.Nachname,
  Mitarbeiter.Telefonnummer, Mitarbeiter.EmailAdresse
    FROM Abteilung
  LEFT JOIN Mitarbeiter
    ON Mitarbeiter.AbteilungID = Abteilung.AbteilungID
WHERE Abteilung.Bezeichnung = 'HW-Entwicklung';
```

Abbildung 2.2: Beispiel sqlQuery - Select mit Join

links zu sehen. Rechts sieht man die Datensätze der beiden Quelltabellen des Querys. Wahrscheinlich würde man diesen einfachen Datenbank Join, in C/C++, mit den Daten in Structures gespeichert, mittels verschachtelter For-Schleifen implementieren. Das Problem bei dieser Implementierung ist, dass man durch jedes Element iterieren muss. Das DBMS hat zur Lösung dieses Problems bessere Algorithmen hinterlegt (Stichwort: binärer Suchbaum, Hash-Maps...).

2.3.3 Stored Procedurs

Ein SQL Server unterstützt nicht nur das Manipulieren und Ausgeben von Daten mit SQL, sondern auch das Speichern von Funktionen und Prozeduren. Diese Prozeduren und Funktionen sind auch in SQL geschrieben. Sie ermöglichen dem Entwickler das Auslagern komplexer SQL Querys. Der wesentliche Unterschied zwischen einer Prozedur und einer Funktion besteht darin, dass eine Funktion einen Rückgabewert haben kann, eine Prozedur dagegen nicht. Das Fehlen eines

Rückgabewerts einer Prozedur stellt aber, entgegen der allgemeinen Erwartung, keine Einschränkung dar. Prozeduren und Funktionen akzeptieren auch sessionbezogene globale Variablen als *OUT* Argument. Desweiteren haben Prozeduren entgegen Funktionen die Möglichkeit, SQL Querys zur Laufzeit zusammenzusetzen und auszuführen. Funktionen können, in SQL Querys benutzt werden, Prozeduren nicht. Prozeduren werden durch ein *CALL* Befehl aufgerufen.

2.4 OPC UA

Open Platform Communications Unified Architecture (OPC UA) ist ein Thema, das immer wieder mit Industrie 4.0 in Verbindung gebracht wird. Luber [1] fasst OPC UA in den wenigen Worten zusammen:
„*OPC UA (Open Platform Communications Unified Architecture) ist eine Sammlung von Standards für die Kommunikation und den Datenaustausch im Umfeld der Industrieautomation. Mithilfe von OPC UA werden sowohl der Transport von Machine-to-Machine-Daten als auch Schnittstellen und die Semantik von Daten beschrieben. Die komplette Architektur ist serviceorientiert aufgebaut.*“
Eine der großen Herausforderungen der Industrie 4.0 besteht in der Vernetzung aller Geräte in einer Fabrik. Dies ist nur durch die Verwendung eines offenen (herstellerunabhängigen) Standards möglich. Ein solcher Standard ist OPC UA. Er definiert nicht nur wie Daten ausgetauscht werden, sondern auch wie die Daten zu interpretieren sind (Semantik). Dabei bedient sich ein OPC UA Server einem eingebauten Informationsmodell. Dies kann man sich vorstellen wie Klassen und Objekte in der objektorientierten Programmierung. Als Beispiel bietet Hersteller A eine Pumpe am Markt an. Diese Pumpe besitzt eine OPC UA Schnittstelle (Server). Dieser Server hat ein Pumpenobjekt mit der Bezeichung „*pumpe_xyz_A*“, das eine Menge an Attributen besitzt die man lesen, schreiben, oder lesen und schreiben kann. Jedes Attribut hat einen eindeutigen Datentyp. Das Attribut *Seriennummer* besitzt zum Beispiel den primitiven Typ *String*. Ein OPC UA Server unterstützt aber nicht nur primitive Datetypen, sondern auch Objekte wie das Pumpenobjekt „*pumpe_xyz_A*“ des Herstellers A. Nun möchte man diese Pumpe des Herstellers A durch ein Programmable Logic Controller (PLC) des Herstellers B auslesen beziehungsweise steuern. Bei einer propriotären Schnittstelle, würde dies zu Problemen führen da dem Hersteller B das Informationsmodell der Pumpe nicht bekannt ist. Hersteller A müsste in diesem Fall Dokumente liefern, mit denen Hersteller B die Pumpe ansteuern könnte. Der Hersteller wäre also gezwungen, für jeden größeren PLC Hersteller, verschiedene Dokumente zur Verfügung zu stellen. OPC UA geht an dieser Stelle einen anderen Weg. Der OPC UA Standard stellt eine Menge an OPC UA Klassen zur Verfügung, von denen die Hersteller nun ihre eigenen Klassen ableiten können. Eine solche Klasse gibt es auch für eine Pumpe. Diese Klasse besitzt nun die minimalen Attribute einer Pumpe. Hersteller A leitet nun ihre Pumpenklasse („*pumpe_xyz_A*“) von

dieser Pumpenklasse ab und erweitert die Klasse um ihre eigenen zusätzlichen Attribute. Wenn jetzt Hersteller B die Pumpe mit ihrem PLC ansprechen möchte, gibt es zwei Möglichkeiten. Es ist bekannt, dass die Pumpenklasse „*pumpe_xyz_A*“ von der allgemeinen Pumpenklasse des Standards abgeleitet wurde. Damit ist es möglich, zumindest die minimalen Funktionen der Pumpe, anzusprechen. Möchte man sich nicht darauf beschränken, sondern den vollen Funktionsumfang nutzen, den die Pumpe bietet, so kann der PLC sich bei dem OPC UA Server nach deren Informationsmodell für die Pumpe erkundigen. Die Attribute sind damit nicht nur in ihrem Typ bekannt, sondern auch in ihrer Bedeutung (Semantik). Als Transportprotokoll benutzt OPC UA das Internet Protocol (IP). Damit ist eine horizontale sowie vertikale Vernetzung innerhalb der Automatisierungspyramide möglich [1]. OPC UA untersützt dabei ein Konzept bei dem Daten angefragt werden (zyklisches lesen), sowie ein Publish/Subscribe Konzept, bei dem man sich Datenknoten abonnieren kann und fortan von dem Server über Änderungen informiert wird [1].

2.5 Echtzeit

Laut Scholz [2] ist Echtzeit wie folgt definiert:
„*Unter Echtzeit versteht man den Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.*“
Das bedeutet, dass ein System nur dann echtzeitfähig ist, wenn es die folgenden Bedingungen erfüllt:

- Jede Komponente des Systems, die an dem Prozess beteiligt ist, muss selbst echtzeitfähig sein.
- Die Rechenzeit der verwendeten Algorithmen muss endlich sein und deren Maximum kalkulierbar.
- Die Transportprotokolle, die beteiligt sind dem Prozess Daten zuzuführen bzw. Daten zu entnehmen, müssen die Echtzeitfähigkeit unterstützen.

Wenn die obere Definition nun auf das Ethernet oder das TCP Protokoll angewand wird, fällt schnell auf, dass diese beide normalerweise nicht echtzeitfähig sind. Ethernet verhindert auf dem Bus (physikalischer Layer) keine Kollisionen sondern erkennt diese nur. Wenn eine Kollision erkannt wurde, wird das Senden auf dem Bus abgebrochen und nach einer zufälligen Zeit erneut versucht. Dieses Verfahren nennt man Carrier Sense Multiple Access/Collision Detection (CSMA/CD). Wer den Zugriff auf den Bus bekommt ist Zufall. Deshalb ist nicht garantiert, dass man innerhalb einer definierten Zeitspanne die Daten

übertragen kann. Man kann dieses Problem allerdings umgehen, indem man im physikalischen Layer bereits dafür sorgt, dass keine Datenkollision auftreten kann. Dies ist zum Beispiel durch den Einsatz von Switches möglich. Diese trennen die einzelnen Sender voneinander und verhindern damit gezielt Kollisionen. Es muss an dieser Stelle angemerkt werden, dass man immernoch nicht vollständig echtzeitfähig Daten übertragen kann, da das Netzwerk auch auf die aufkommende Datenlast ausgelegt sein muss. Dadurch ist es möglich mit Ethernet in einer kontrollierten Umgebung Daten in Echtzeit zu übertragen. Ein weitverbreiteter Irrtum ist, dass Echtzeitfähigkeit bedeutet etwas müsse besonders schnell reagieren, es genügt dass die Reaktionszeit definiert und endlich ist.

3 Systementwurf

3.1 Use-Cases

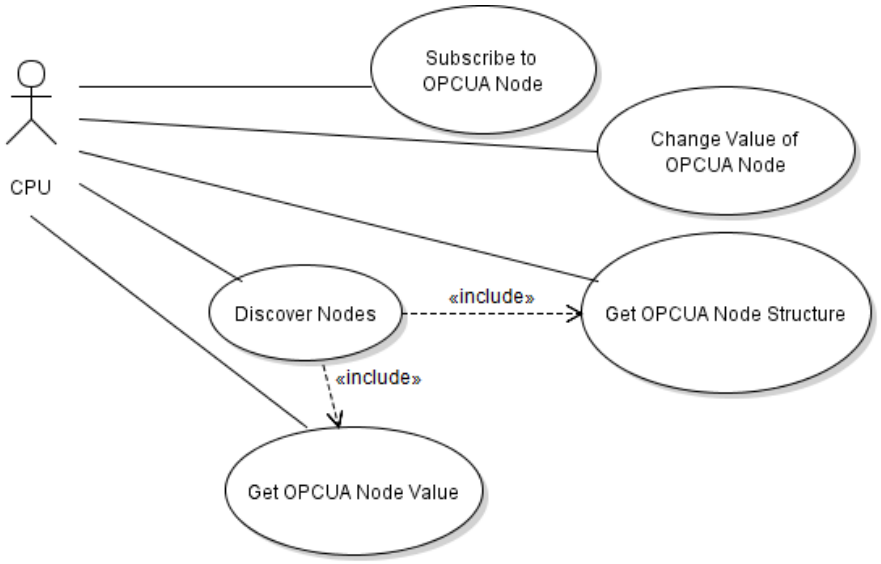


Abbildung 3.1: Use-Case Diagramm des SCADA Systems II

Aus der Zielsetzung der Arbeit ergeben sich die Anwendungsfalldiagramme in Abbildungen 3.2 und 3.1. Das Diagramm in Abbildung 3.2 beschreibt abstrakt die Anwendungsfälle, die das SCADA System für den Anwender, in der Rolle des Bedieners (Operator), sowie des Programmierers (Programmer) erfüllen muss. Analog dazu beschreibt das Anwendungsfalldiagramm in Abbildung 3.1 die Anwendungsfälle die ein PLC an das SCADA System stellt. Es ist möglich dies in zwei getrennten Diagrammen abzuhandeln, da es keine Anwendungsfälle gibt, die einen Akteur aus beiden Diagrammen benötigt. Jedoch interagieren alle Akteure mit dem selben System.

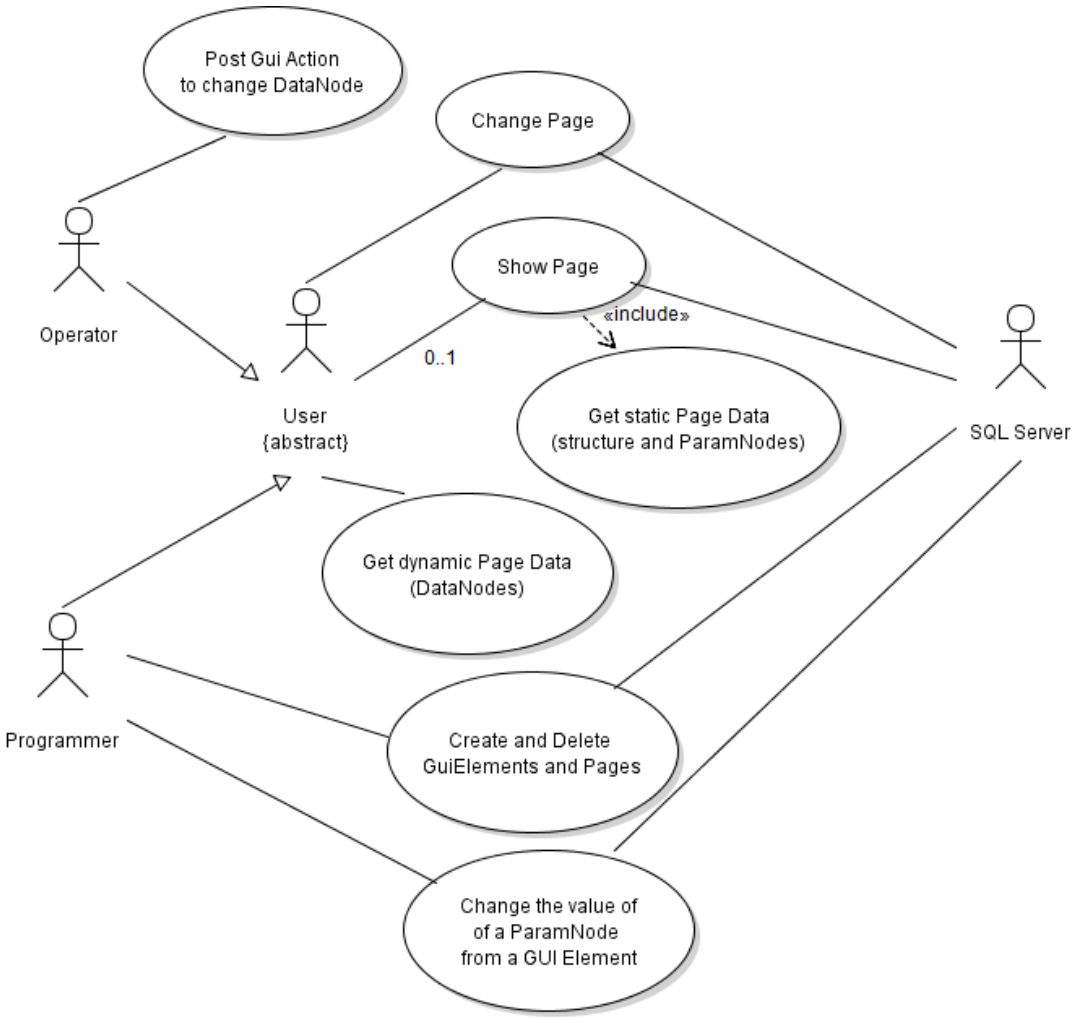


Abbildung 3.2: Anwendungsfalldiagramm des SCADA Systems I

3.2 Architektur

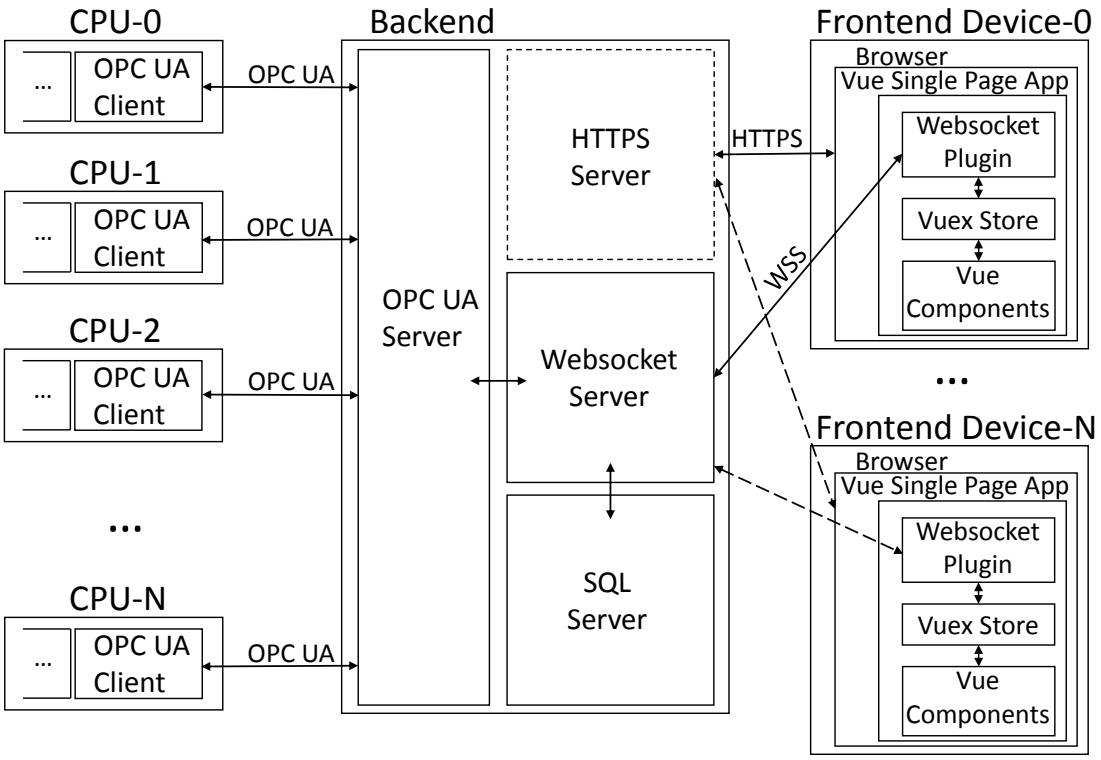


Abbildung 3.3: Kommunikationsmodell des SCADA Systems.
Die CPUs sind per OPC UA an das Scada System angebunden und haben die Rolle eines OPC UA Clients. Die Webapplikation im Browser kommuniziert über WSS sowie HTTPS verschlüsselt mit dem Backend.

Wie in Abbildung 3.3 dargestellt, lässt sich die Architektur des SCADA Systems in PLCs, Backend und Frontends unterteilen. Unter Frontend (Abschnitt 3.2.2) versteht man die (in diesem Fall) grafische Schnittstelle, die es dem Nutzer ermöglicht mit dem System zu interagieren. Das Backend (Abschnitt 3.2.1) fasst den Rest des Systems zusammen.

3.2.1 Backend

Das Backend besteht aus den folgenden vier Komponenten:

- Einem Hypertext Transfer Protocol Secure (HTTPS) Server
- Einem WebSocket Secure (WSS) Server
- Einem SQL Server

- Einem OPC UA Server

Damit ergeben sich, wenn man das Backend von außen betrachtet, als Schnittstellen OPC UA, WSS und HTTPS. Über diese Schnittstellen stellt das Backend den PLCs, sowie dem Frontend, Services zur Verfügung. Eine Sonderrolle unter den Komponenten nimmt dabei der HTTPS Server ein, da er als einzige Komponente keine Verbindung mit dem Rest des Backends hat. Dies ist deshalb möglich, da der HTTPS Server dazu da ist, die Hypertext Markup Language (HTML), die Cascading Style Sheets (CSS), sowie die JavaScript (JS) Dokumente einmalig beim Seitenaufruf an das Endgerät (Frontend Host Device) auszuliefern. Von diesem Zeitpunkt an findet die Kommunikation zwischen Frontend und Backend ausschließlich über den WSS Server statt. Dabei exist immer genau eine Session für jede Frontendinstanz.Zwischen dem Websocket Server und dem Frontend können ab diesem Zeitpunkt Strings ausgetauscht werden. Der OPC UA Server hält die variablen Daten des Systems und bietet diese den angeschlossenen PLCs an. Die Parametrierung des Frontends sowie alle anderen Daten, die persistiert werden müssen, werden in einem SQL Server, in Form einer relationalen Datenbank, gespeichert.

3.2.2 Frontend

Das Frontend ist eine Webapplikation, mit Vue.js als Webframework.
Beim Laden der Seite wird eine WSS Verbindung zum Backend aufgebaut. Ohne Authentifizierung zeigt das Frontend eine Seite an, die es dem Nutzer ermöglicht sich durch individuelle Zugansdaten (Benutzername und Passwort) zu authentifizieren. Werden auf dieser Seite gültige Zugangsdaten eingegeben, ist die WSS Session authentifiziert und zeigt das Layout in Abbildung 3.4 an. Das Layout beinhaltet oben links einen Button um die Navigation der Seite ein und auszublenden. Ist die Navigation am linken Rand eingeblendet, so ermöglicht sie es zwischen den einzelnen Seiten der Visualisierung zu navigieren. Dazu zeigt die Navigation immer einen „Zurück“ Button, um in die übergeordnete Seite zu wechseln, sowie Buttons um zu den untergeordneten Seiten zu wechseln. Existieren keine untergeordneten Seiten, oder keine übergeordnete Seite, so existieren auch die Buttons nicht. Schließlich wird im zentralen Fenster die aktuelle Page der Visualisierung angezeigt. Der komplette Seiteninhalt ist eine Repräsentation des Datenobjekts, das global in der Webapplikation vorliegt. Diese Datenstruktur ist eine Kopie der Daten aus dem Backend (Abschnitt 3.3.1), welche die aktuell angezeigte Seite betreffen. Sie ist in Abschnitt 3.3.2 genauer beschrieben und kann nur über die Websocket Schnittstelle verändert werden. So ist sichergestellt, dass die GUI Elemente immer die Daten des Backends darstellen und es keine Unterschiede gibt. Der Zustand ist also über alle Instanzen des Backends und Frontends immer konsistent. Wie Daten verändert werden können ist genauer in Abschnitt 3.2.3 beschrieben. Innerhalb der Webapplikation existiert eine Page

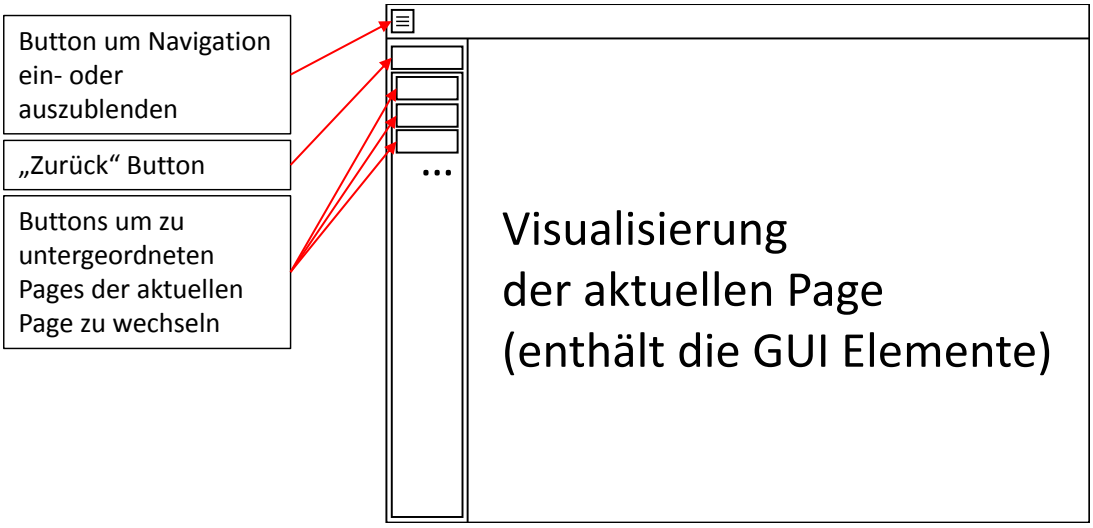


Abbildung 3.4: Layout des Frontends bei einer authentifizierten WSS Session

Komponente die, entsprechend der GUI Elemente in der globalen Datenstruktur, GUI Element Komponenten (*guiElement*) instanziert und auf der Page anzeigt. Diese GUI Elemente werden entsprechend ihres Typs (zum Beispiel *Button*) weiter in spezialisierte GUI Elemente (z.B. *guiElementButton*) unterteilt. Das Schema, wie innerhalb der Webapplikation auf Daten zugegriffen werden kann, bzw. Daten geändert werden können, ist in Abbildung 3.5 dargestellt. Die GUI Elemente haben nun Zugriff auf eine Funktion um Datenänderungen anzufordern, sowie auf den ihnen zugeordneten Teil der globalen Datenstruktur. Diese Datenstruktur ist in dem Vue Plugin *VueX* gespeichert und existiert damit nur einmal pro Webapplikation. Wie der Dispatcher die Struktur des Frontends ändert ist

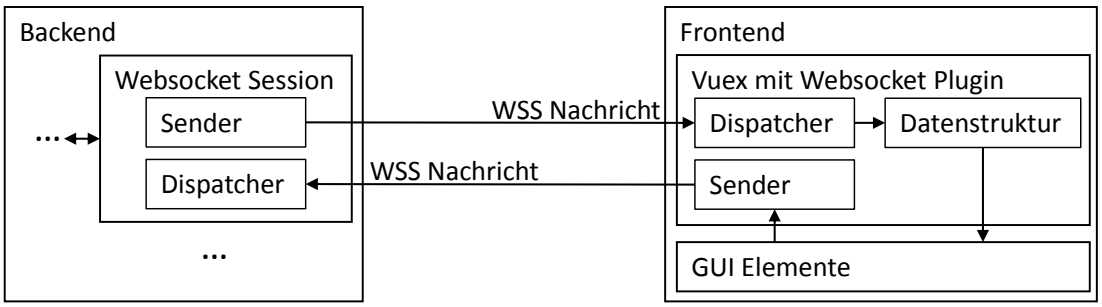


Abbildung 3.5: Schema des Zugriffs auf die Datenstruktur des Frontends

in Abschnitt 3.2.3 beschrieben. Die GUI Elemente können nur lesend auf die Datenstruktur zugreifen.

3.2.3 Protokoll zwischen Frontend und Backend

Websockets bieten die Möglichkeit asynchron Daten zwischen Frontend und Backend auszutauschen, bieten allerdings keinerlei Regeln für die Semantik dieser Daten. Deshalb ist ein weiteres Protokoll in der Applikationsebene erforderlich, welches die Semantik der ausgetauschten Daten definiert. An dieses Protokoll werden folgende Anforderungen gestellt.
Das Protokoll soll

... so einfach wie möglich sein.

... alle Datentypen unterstützen, die das Backend definiert.

... weitestgehend symmetrisch sein (Die Nachrichten vom Server zum Client haben eine ähnliche Gestalt wie die vom Client zum Server).

... möglichst leichtgewichtig sein.

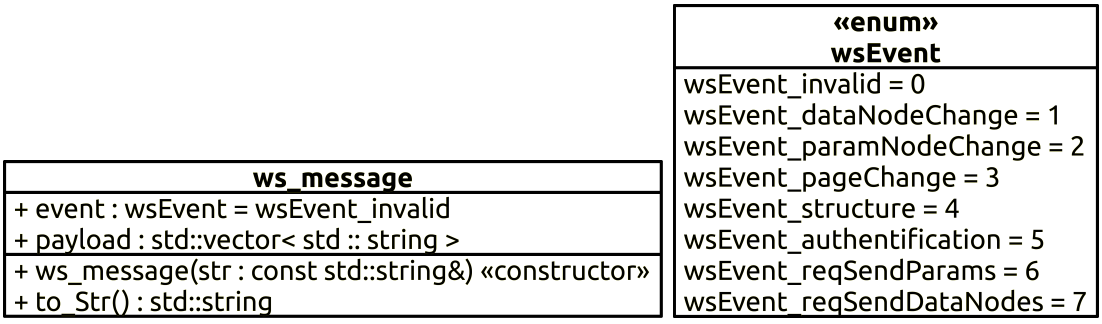
... möglichst frei von Zuständen sein, sodass ein Vertauschen zweier Nachrichten keine Relevanz für die Applikation darstellt.

... eine Authentifizierung des Nutzers ermöglichen.

Die meisten Anforderungen können erfüllt werden, allerdings stehen die letzten drei in Konkurrenz zueinander. Um eine Authentifizierung zu ermöglichen gibt es, bei einer Verbindung die ab dem Aufbau dauerhaft besteht, folgende Möglichkeiten:

- Die Zugangsdaten, die die Session authentifizieren, werden einmalig vom Client zum Server gesendet. Die Session ist dann authentifiziert und hat ab diesem Zeitpunkt mehr Rechte.
- Die Zugangsdaten werden bei jeder Nachricht mitgesendet und überprüft.

Die erste Variante ist möglichst leichtgewichtig, da die Zugangsdaten nur einmal gesendet werden. Allerdings ist sie nicht frei von Zuständen, da die Schnittstelle, je nachdem ob eine Session bereits authentifiziert ist, unterschiedlich auf eingehende Nachrichten reagiert. Die zweite Variante reagiert immer identisch (deterministisch) auf die gleiche Nachricht, ist aber weniger performant, da das Verhältnis von Protokoll-Overhead zu Nutzdaten immer größer ist als bei der ersten Variante. Aufgrund dieses Sachverhalts wird bei der Zustandsfreiheit eine Ausnahme gemacht werden.
Das erarbeitete Protokoll tauscht Strings aus, dessen Struktur der Klasse in Abbildung 3.6a zugrunde liegt. Diese Klasse besteht aus Nutzdaten (*payload*), sowie einem Header (*event*). Nutzdaten sind nur optional vorhanden, so ist möglich nur ein Event zu verschicken. Der Header ist eine Aufzählung verschiedener Events.



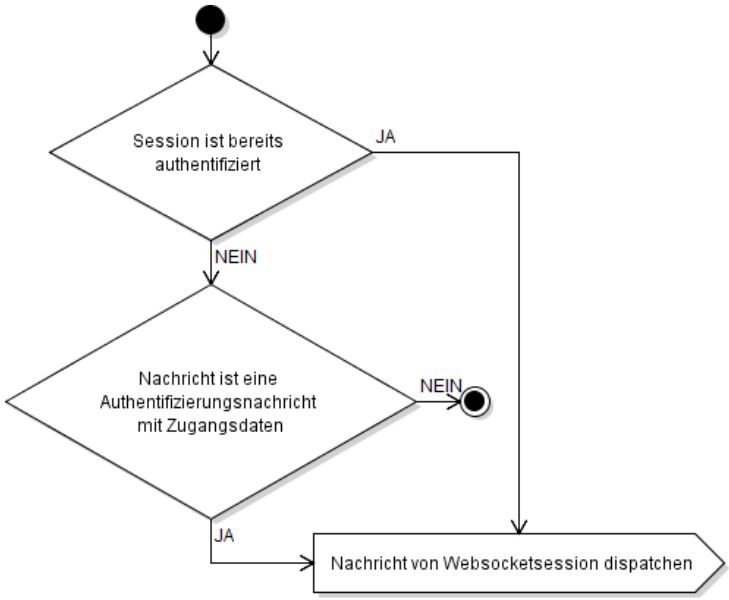
(a) Websocket Message Container mit *event* (Header) und *payload* (Nutzdaten)

(b) Websocket Event Definition als Aufzählung

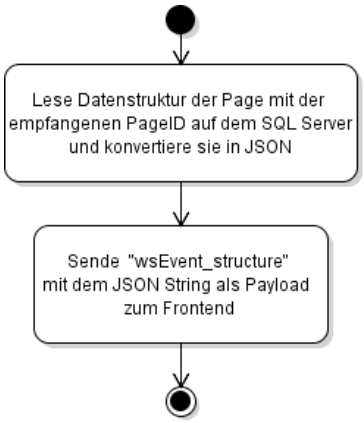
Abbildung 3.6: Websocket Message Container und Event

Diese Aufzählung (*wsEvent*) ist in Abbildung 3.6b dargestellt und gibt die einzelnen Typen von Nachrichten an. Ein solches Objekt kann seitens des Backends aus dem String in einer Websocket Nachricht konstruiert, sowie in einen solchen String übersetzt werden. Die Bildungsvorschrift ist dabei, alle Information welche das *ws_message* Objekt enthält (*event* und *payload*), als String zu kodieren und durch ein Semikolon getrennt zu einem String zusammenzusetzen. Dabei stellt das erste Feld in einem versendeten String immer das kodierte Event dar. Die restlichen Felder entsprechen dem Nutzdatenvektor der *ws_message*. Diese Bildungsvorschrift ist umkehrbar, solange der Nutzdatenvektor kein String mit einem Semikolon enthält. Dieser Fall ist unbedingt zu vermeiden und muss vom Frontend und Backend beim Kodieren des Strings abgefangen werden. Die umgekehrte Bildungsvorschrift implementiert die *ws_message* Klasse als Konstruktor. Das Protokoll unterstützt nun die in Abbildung 3.6b dargestellten Events. Das erste Event (*wsEvent_invalid*) ist der Standartwert des Events in jedem *ws_message* Objekt. Auf eine Nachricht mit diesem Event wird beim Empfang von keiner Seite aus reagiert und die Nachricht verworfen. Dies ist eine Absicherung, um zu verhindern, dass eine Nachricht mit einem zufälligen Event versendet wird. Front- sowie Backend stellen einen Dispatcher zur Verfügung welcher, entsprechend des Events einer empfangenen Nachricht, einen Handler aufruft. Der Dispacher des Backends unterscheidet sich vom Dispatcher im Frontend, da er die Authentifizierung des Nutzers realisiert. Das Verhalten, ob und wie eine Nachricht vom Dispatcher im Backend dispatcht wird, ist durch das Aktivitätsdiagramm in Abbildung 3.7a definiert. Hier werden nur Nachrichten dispatcht, wenn die Websocketsession bereits authentifiziert wurde oder die Nachricht eine Authentifizierungsnachricht ist. Ist dies der Fall, werden die entsprechenden Handler zu den empfangenen Events aufgerufen.
Die Handler des Backends sind durch die verbleibenden Diagramme in Abbildung 3.7, sowie durch die Diagramme in Abbildung 3.8 definiert.
Das Frontend bedarf keines solchen Authentifizierungsmechanismus, da dies be-

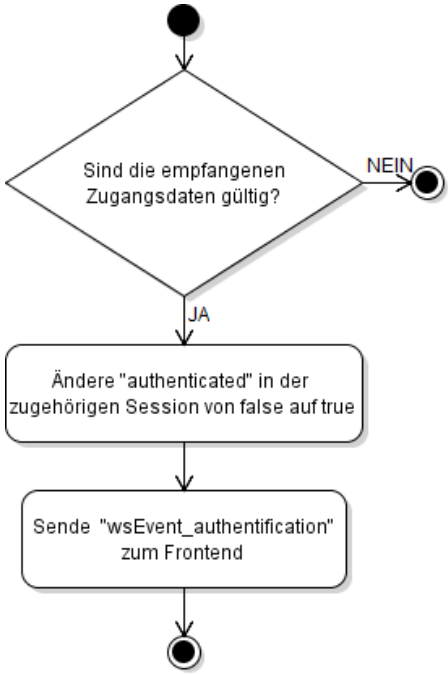
reits im WSS Protokoll implementiert ist. Wie auch bei HTTPS geschieht das durch ein Serverzertifikat des Backends, dass durch eine externe Certification Authority (CA) beim Verbindungsaufbau verifiziert werden muss. Die Handler des Frontends sind in Abbildung 3.9 dargestellt.



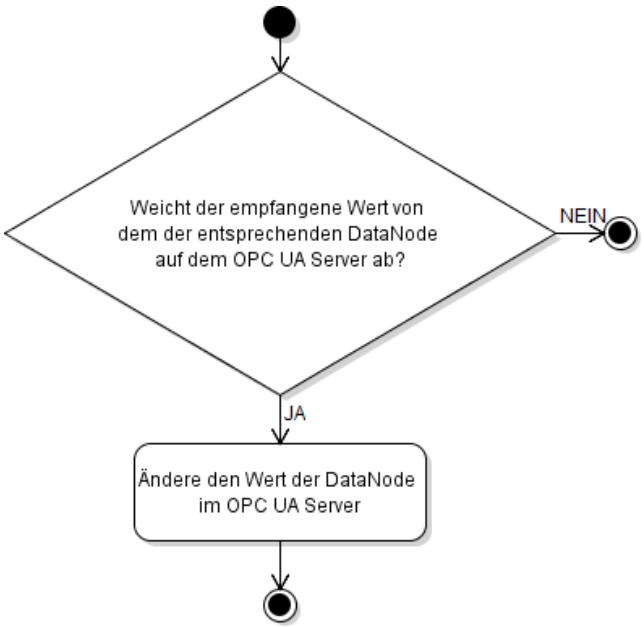
(a) WebsocketEvent empfangen



(b) WebsocketEvent: *Structure* empfangen

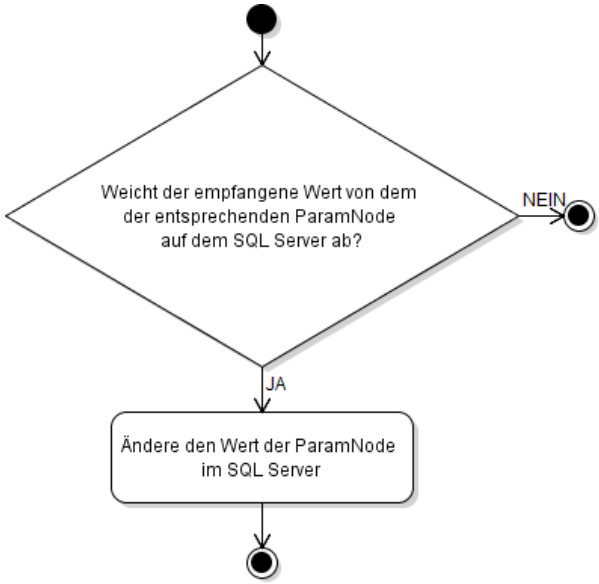


(c) WebsocketEvent: *Authentification* empfangen

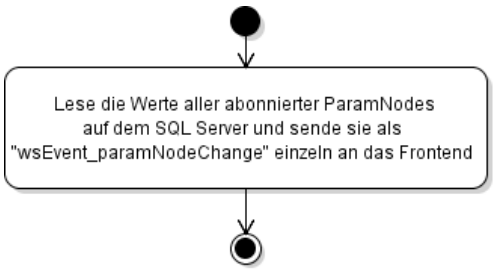


(d) WebsocketEvent: *DataNodeChange* empfangen

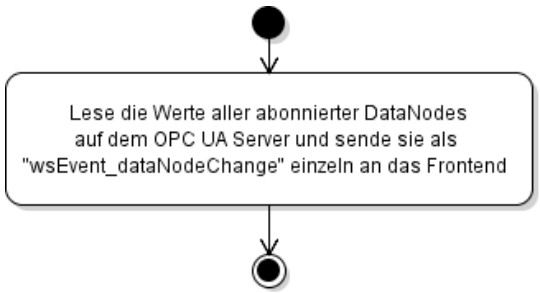
Abbildung 3.7: Aktivitätsdiagramme des Dispatchers im Backend I



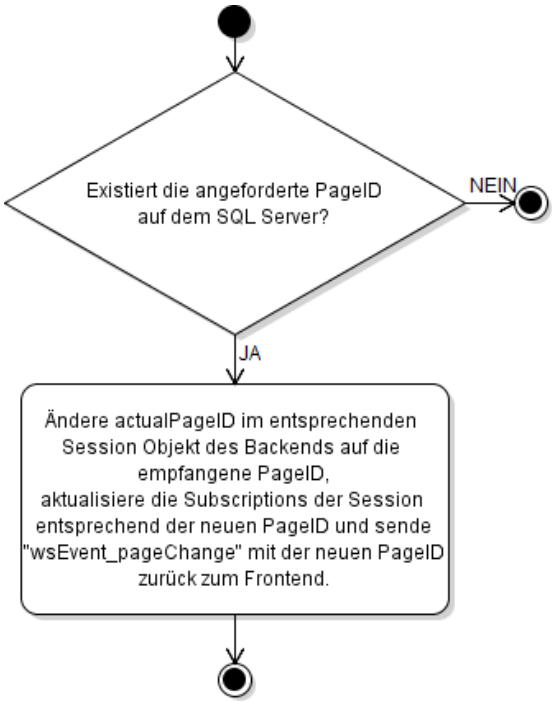
(a) WebsocketEvent: *ParamNodeChange* empfangen



(b) WebsocketEvent: *ReqSendParamNodes* empfangen

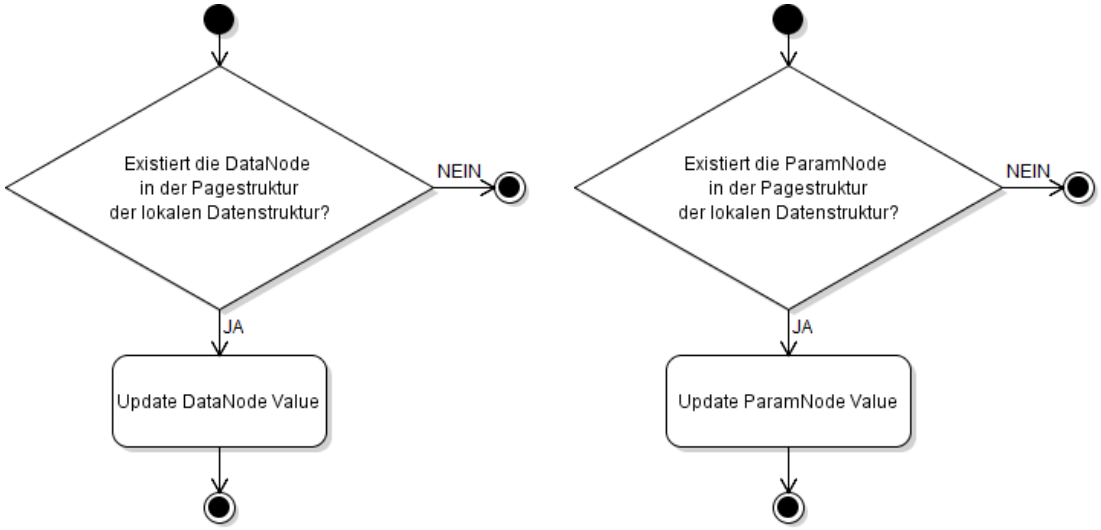


(c) WebsocketEvent: *ReqSendDataNodes* empfangen



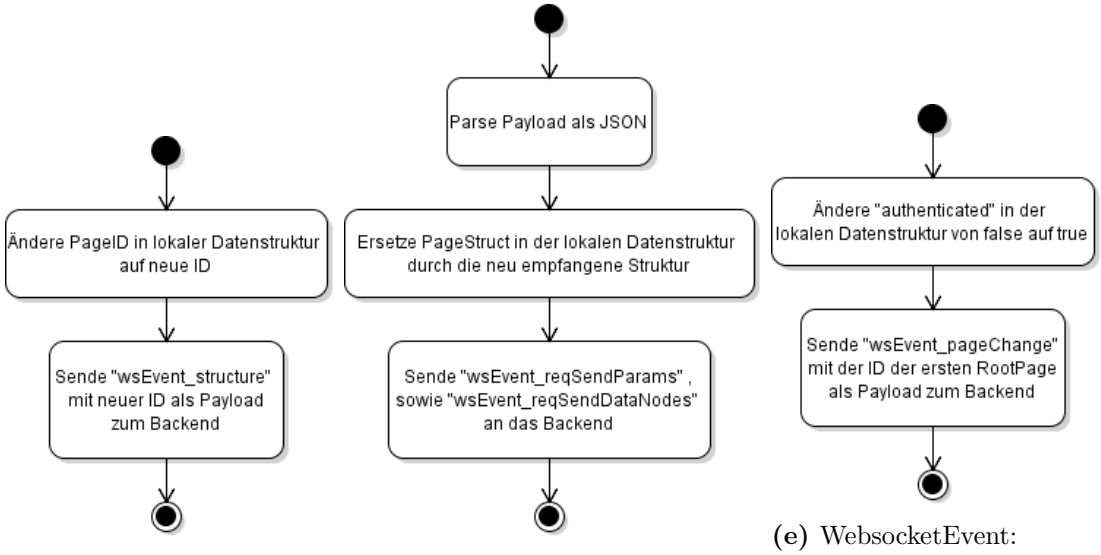
(d) WebsocketEvent: *PageChange* empfangen

Abbildung 3.8: Aktivitätsdiagramme des Dispatchers im Backend II



(a) WebsocketEvent: *DataNodeChange* empfangen

(b) WebsocketEvent: *ParamNodeChange* empfangen



(c) WebsocketEvent: *PageChange* empfangen

(d) WebsocketEvent: *Structure* empfangen

(e) WebsocketEvent: *Authentification* empfangen

Abbildung 3.9: Aktivitätsdiagramme des Dispatchers im Frontend

3.3 Datenstruktur

3.3.1 Backend

Das abgebildete Entity Relationship Diagram (ERD) in Abbildung 3.10, stellt die globale Datenstruktur der Datenbank auf dem SQL Server dar. Sie besteht aus einzelnen Tabellen deren Primärschlüssel, ausgenommen einzelner Hilfstabellen, immer eine ID ist. Zentrales Element sind dabei die GUI Elemente (Tabelle *GuiElements*). Jedes GUI Element ist einer Page (Tabelle *Pages*) zugeordnet. Eine Page kann wiederum einer anderen Page zugeordnet sein. Dies wird erreicht indem jeder Page-Datensatz eine ParentID enthält, welche als Fremdschlüssel auf die übergeordnete Page zeigt. So entsteht eine Baumstruktur innerhalb der Entitäten. Jedes GUI Element hat n Parameter und m Data Nodes (mit $n, m \in \mathbb{N}$). Diese werden in den Tabellen *GuiElementParams* sowie *GuiElementDataNodes* verwaltet. Jede DataNode, beziehungsweise jeder Parameter, hat einen Datentyp, einen Namen, eine Beschreibung, sowie einen Wert. Bei den Parametern entspricht dies dem tatsächlichen aktuellen Wert, bei den DataNodes entspricht dies dem initialen Wert beim Starten des Backends. Die zuvor beschriebenen GUI Elemente besitzen außerdem einen Typ (verwaltet in der Tabelle *GuiElementTypes*). Jedem Typ sind nun n DataNodeTemplates (Tabelle *GuiElementDataNodeTemplates*) zugeordnet, aus denen die eigentlichen DataNodes beim Instanzieren erstellt werden. Da eine solche Vorlage für DataNodes auch für mehrere GUI Element Typen verwendet werden kann, ist die Verbindung eine $n : m$ Bindung. Dies ist in einer Datenbank nur durch eine Hilfstabelle, welche die Primärschlüssel zweier Tabellen miteinander verknüpft, ereichbar. Analog zu den DataNodes ist der gleiche Mechanismus für die Parameter vorgesehen. Um ein sauberes Interface zu schaffen sind zum Instanzieren, sowie zum Löschen der GUI Elemente, Stored Procedures vorgesehen. Die komplette Datenstruktur des SQL Servers ist so abgelegt, dass dieser alle relevanten Relationen kennt und entsprechend verwaltet. So werden beispielsweise beim Löschen eines GUI Elements alle zugehörigen DataNodes mitgelöscht und beim Löschen einer kompletten Page werden alle der Page zugeordneten GUI Elemente auch gelöscht (*on delete cascade*, Abschnitt 2.3.1).
Der OPC UA Server hält alle DataNodes der GUI Elemente. Diese GUI Elemente werden entsprechen der Pages, denen sie zugeordnet sind, auf dem OPC UA Server abgelegt und sind den PLCs dort zugänglich. Ob eine DataNode von den PLCs nur gelesen oder auch beschrieben werden kann, wird durch das Flag *writePermission* auf dem SQL Server in der *DataNodeTemplate* Tabelle festgelegt. Beim Erstellen der DataNode auf dem OPC UA Server, wird dieses Flag ausgelesen und direkt in die Konfiguration des OPC UA Servers übernommen. Der native Datentyp, mit dem eine DataNode auf dem OPC UA Server abgelegt wird, wird den DataNodeTemplates des SQL Servers entnommen.
Auf dem SQL Server haben die DataNodes, sowie Parameter, immer den nativen

SQL Datentyp *VARCHAR*, um ein Speicherfeld zu erhalten, das alle benutzten OPC UA Datentypen unterstützt. Beim Lesen, beziehungsweise beim Schreiben, wird der als String gespeicherte Wert auf dem SQL Server, entsprechend seines explizit abgespeicherten Datentyps, konvertiert. GuiElements und Pages haben als Datentyp auf dem OPC UA Server den Typ BaseObjectType. Auf einen Spezialisierung dieser Typdefinition für Pages und GUI Elements auf dem OPC UA Server wird absichtlich verzichtet, da sie nur zur Strukturierung der DataNodes, auf dem OPC UA Server, genutzt werden.

3.3.2 Frontend

Die Datenstruktur des Frontends wird in dem Vue Plugin *VueX* gespeichert.

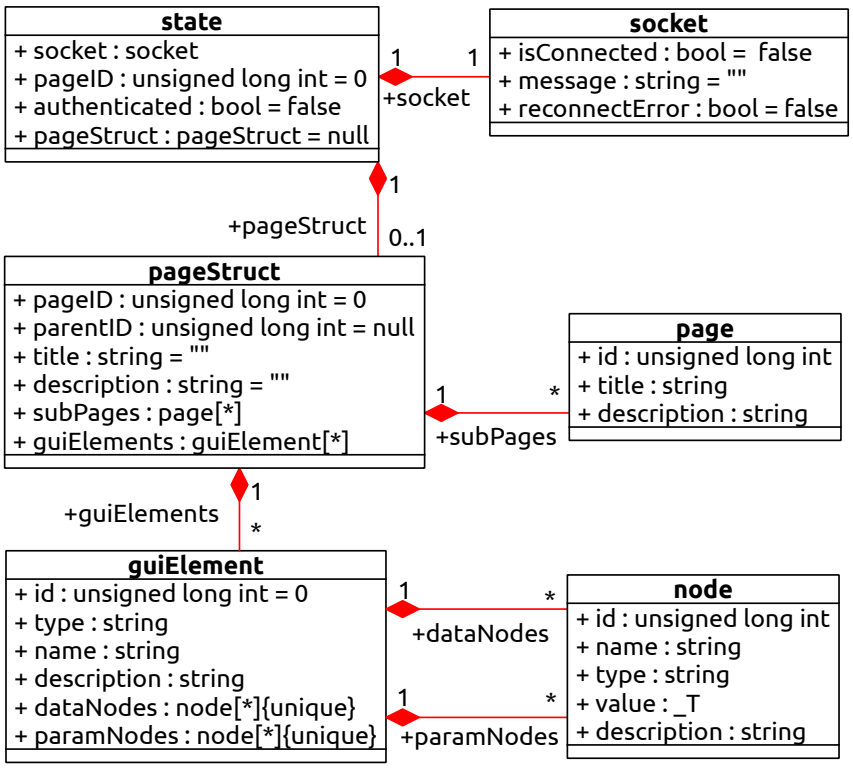


Abbildung 3.11: Frontend Datenstruktur

Diese Struktur ist in Abbildung 3.11 als Klassendiagramm dargestellt. Sie existiert nur einmal global im Frontend. Dabei macht VueX die Struktur, die im „*state*“ Objekt gespeichert ist, der restlichen Applikation verfügbar. Die angegebenen Datentypen im Diagramm geben nicht die nativen JavaScript Datentypen an, sondern die Typen, die im C++ Backend verwendet werden und als JavaScript Object Notation (JSON) an das Frontend übermittelt werden. Die real im Frontend verwendeten Datentypen sind weniger spezifisch, da JavaScript zum Beispiel

alle Zahlen unter dem Typ „*Number*“ abspeichert und dabei keine Unterscheidung des Typs unterstützt. Das „*state*“ Objekt enthält ein „*socket*“ Objekt, in dem die Daten der Websocket Verbindung gespeichert sind. Zusätzlich speichert es die „*pageID*“, die angibt welche Seite aktuell dargestellt wird, eine Variable „*authenticated*“, als Indikator ob die Websocket Verbindung bereits authentifiziert ist, sowie ein „*pageStruct*“ Objekt, das alle weiteren Daten zur aktuellen Seite beinhaltet. Wird noch keine programmierte Seite des SCADA Systems angezeigt, dann ist die Variable „*pageStruct*“ *null* und es ist keine „*pageStruct*“ vorhanden. Das ist im Diagramm durch die explizite Angabe der Multiplizität (1 : 0..1) bei der Komposition gekennzeichnet. Bei der Initialisierung des „*state*“ Objekts ist das „*pageStruct*“ Objekt immer *null*, bis es gemäß des Protokolls aus Abschnitt 3.2.3 als JSON empfangen wird. Das „*pageStruct*“ Objekt wird aus den Daten des Backends gebildet, welche eine einzelne Seite des SCADA Systems betreffen. Es enthält die „*parentID*“ der übergeordneten Seite. Ist die dargestellte Seite die Einstiegsseite und hat damit keine übergeordnete Seite, so ist diese Variable *null*. Zusätzlich enthält sie ein Array mit Objekten, welche die wesentlichen Information zu den untergeordneten Seiten beinhalten. Diese Objekte wurden in dem Klassendiagramm als „*page*“ Klasse gekennzeichnet und beinhalten die ID der Seite („*id*“), den Titel („*title*“) sowie die Beschreibung („*description*“). Diese Informationen werden im Frontend benötigt, um eine Navigation durch die einzelnen Seiten des SCADA Systems zu ermöglichen. Die restlichen Attribute des „*pageStruct*“ Objekts stellen die Daten der aktuell angezeigten Seite dar. Wie auch in der „*page*“ Klasse sind das die ID, der Titel und die Beschreibung. Zusätzlich enthält das „*pageStruct*“ Objekt beliebig viele GuiElemente („*guiElements*“). Diese werden ae werden allerdings nicht als Array gespeichert, sondern als Objekt, indem jedes Attribut des Objekts ein „*guiElement*“ Objekt ist. Die Attributnamen sind dabei die IDs der guiElemente, sodass ein Zugriff auf ein einzelnes Element über deren ID möglich ist. Jedes „*guiElement*“ besteht aus einer ID („*id*“), einem Typ („*type*“), einem Namen („*name*“), einer Beschreibung („*description*“), sowie so vielen DataNodes und ParamNodes wie die Vorlage der Backend Datenstruktur (Abbildung 3.10), des entsprechenden „*GuiElementTypes*“, vorgibt. Die Nodes werden dabei in den zwei Attributen „*dataNodes*“ sowie „*paramNodes*“ als Objekt gespeichert. Die Attributnamen bilden dabei die Namen der Nodes, da diese pro „*guiElement*“ Objekt nur einmal vorhanden sind. Jedes „*node*“ Objekt hat, unabhängig ob es eine „*dataNode*“ oder eine „*paramNode*“ repräsentiert, immer eine ID („*id*“), einen Namen („*name*“), einen Typ („*type*“) als String, eine Wert („*value*“) sowie eine Beschreibung („*description*“). Ob eine Node eine „*DataNode*“ (Datenpunkte vom OPC UA Server) oder eine „*ParamNode*“ (auf SQL Server) ist, ist nur durch ihre Zugehörigkeit zu dem entsprechenden Objekt des Attributs „*dataNodes*“ oder „*paramNodes*“, im „*guiElement*“ Objekt, definiert. Der Typ einer Node ist immer der Typ, entsprechend der expliziten Angabe des Typs (Attribut „*type*“), eines einzelnen „*node*“ Objekts.

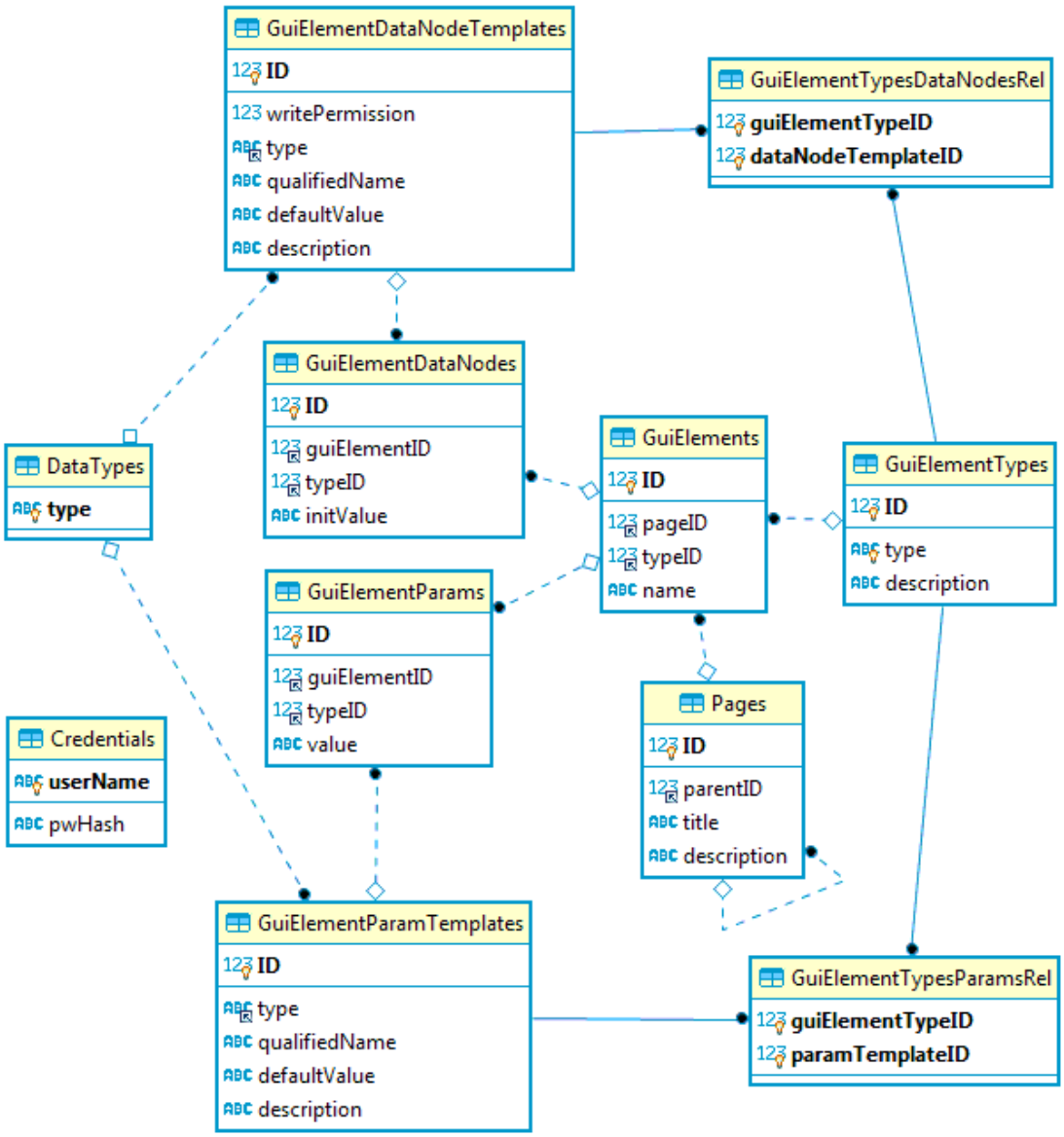


Abbildung 3.10: ERD des SCADA Systems.
Darstellung zeigt die einzelnen Tabellen der Datenbank auf dem SQL Server und deren Beziehungen zueinander.

4 Umsetzung des Proof of Concept

Ein PoC ist eine Möglichkeit um die Realisierbarkeit des Systementwurfs aus Kapitel 3 zu belegen. In diesem PoC wird sich auf die wesentliche Funktionalität eines SCADA Systems beschränkt. Dabei wird anhand der Architektur in Abschnitt 3.2 eine Webapplikation auf Basis von *Vue.js* als Frontend, sowie eine ausführbare Anwendung als Backend in C++ programmiert und anschließend zusammen mit einem Webserver und einem SQL Server getestet. Die Dokumentation dieses PoCs ist in den folgenden zwei Abschnitten 4.1 und 4.2 enthalten.

4.1 Backend

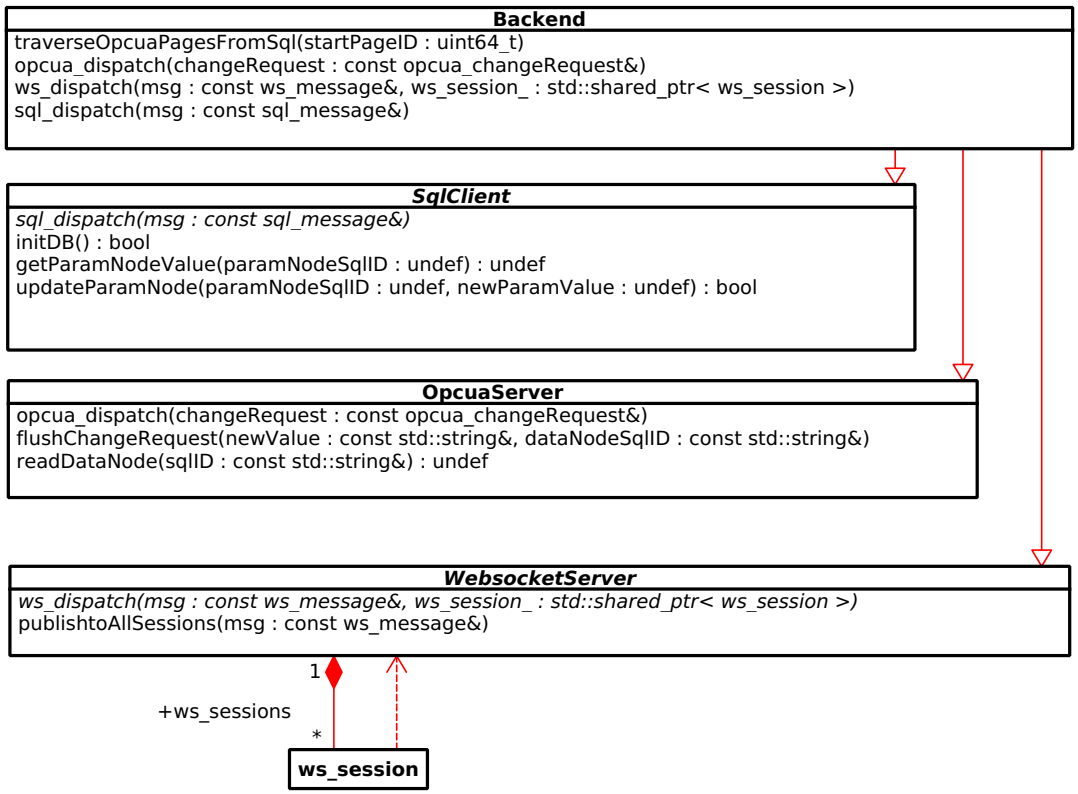


Abbildung 4.1: Klassendiagramm Überblick Backend

Das Klassendiagramm in Abbildung 4.1 stellt die Grundstruktur des Backends

dar. Das Diagramm ist, bis auf die „*Backend*“ Klasse, nicht vollständig und auf die wesentlichen Operationen zur Kommunikation reduziert. Die „*Backend*“ Klasse (abgeleitete Klasse) erbt von den drei Basisklassen „*SqlClient*“, „*OpcuaServer*“ und „*WebsocketServer*“. Jede Basisklasse hat ihre eigene dedizierte Aufgabe die in den einzelnen jeweiligen, gleichnamigen Abschnitten genauer beschrieben ist. Die abgeleitete Klasse steht dabei als Vermittler dazwischen. Möchte eine der Basisklassen etwas ausführen, für das eine der anderen Klassen benötigt wird, so kann diese das anstoßen, indem sie eine virtuelle (genauer: „*pure virtual*“) Methode aufruft, die in der Basisklasse selbst nicht implementiert wird, sondern in der abgeleiteten „*Backend*“ Klasse. Dies ermöglicht es nun indirekt durch eine Basisklasse eine Methode einer anderen Basisklasse aufzurufen. Um zu garantieren, dass dies auch geschieht und nicht durch einen Tippfehler eine falsche Methode aufgerufen wird, ist die Methode in der Basisklasse mit „ *= 0* “ definiert, was den Versuch der Dereferenzierung dieser Methode verhindert und mit einem Kompilerfehler kennzeichnet. Diese Eigenschaft einer Methode ist im Klassendiagramm durch die kursive Schrift der Operation gekennzeichnet. In der abgeleiteten Klasse („*Backend*“) wird die dort reimplementierte Methode nun mit dem Schlüsselwort „*override*“ am Ende der Deklaration gekennzeichnet. Dies zwingt den Kompiler zu überprüfen, ob die dort deklarierte Methode wirklich eine Methode der Basisklasse reimplementiert und nicht eine neue Methode ist. Zusätzlich zu den virtuellen Methoden, die in diesem Fall eine Kommunikation zu anderen Basisklassen, indirekt über die abgeleitete Klasse, ermöglichen, existieren noch eine Anzahl an Methoden, über die man mit der Klasse selbst interagieren kann. Die Methode „*traverseOpcuaPagesFromSql(startPageID)*“ erstellt alle notwendigen OPC UA Nodes, auf Basis der Datenstruktur, ab der angegebenen „*startPageID*“.
Eine Sonderrolle nehmen dabei die Operationen ein, die im Klassendiagramm von Abbildung 4.1, außer der bereits beschriebenen noch zusätzlich dargestellt sind. Sie ermöglichen die Änderung oder Abfrage von Daten der jeweiligen Klassen.
Die einzelnen Basisklassen sind zusätzlich separat in den Abbildungen 4.2, 4.5 sowie 4.7 vollständig, mit allen Attributen und Operationen, dargestellt.

4.1.1 WebsocketServer

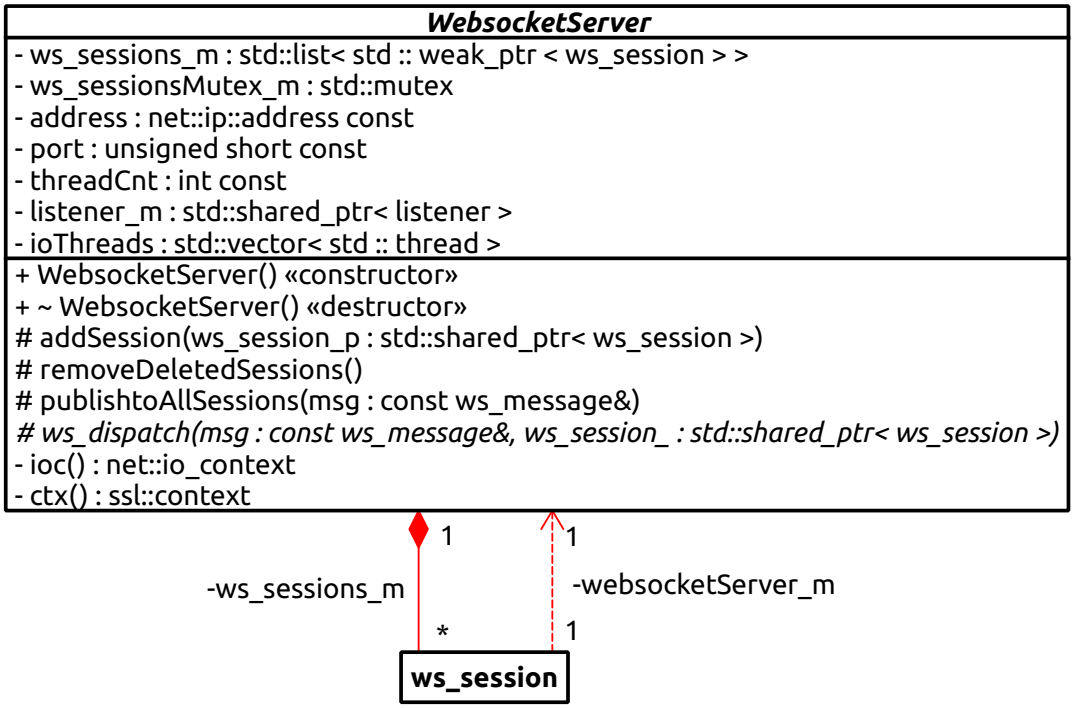


Abbildung 4.2: Klassediagramm der Klasse „*WebsocketServer*“

Die Klasse „*WebsocketServer*“ (Abbildung 4.2) verwaltet die offenen Websocket Sessions (Objekte der Klasse „*ws_session*“). Sie bedient sich dabei der bekannten „*Boost.Beast*“ C++ Bibliothek. Websocket Sessions sind die einzelnen Verbindungen zwischen Backend und Frontend. Das Aufbauen von Verbindungen ist den Beispielen von Vinnie Falco zur „*Boost.Beast*“ Bibliothek entnommen. Die „*WebsocketServer*“ Klasse speichert dabei, pro aktiver Websocket Session, einen Pointer des zugehörigen „*ws_session*“ Objekts in einer Liste. Diese Liste existiert als das Attribut „*ws_sessions_m*“ in der Klasse „*WebsocketServer*“ und ist zusätzlich durch ein Mutex geschützt. Diese Mutex kann man sich vorstellen wie ein Schloss, das immer zuerst entsperrt werden muss, bevor die Session Liste verfügbar ist (Aufruf von „*std::mutex::lock()*“). Ist das Mutex von einem Thread entsperrt, so kann ein anderer Thread es solange nicht entsperren wie es gesperrt ist. Damit wird der zeitgleiche Zugriff von mehreren Threads auf eine Datenstruktur verhindert. Ist ein Thread mit der Arbeit an der geschützten Ressource fertig, gibt er sie wieder frei indem er die Methode „*std::mutex::unlock()*“ aufruft. Das scheint der einfachste Weg zu sein, aber nicht der sicherste. Wenn im Code zwischen „*lock()*“ und „*unlock()*“ eine Exception Eintritt, wird zwar die Ressource danach nichtmehr benötigt, es wird aber auch nicht „*unlock()*“ aufgerufen. Man spricht in diesem Fall von einer *Deadlock* da das Mutex von einem Thread

entsperrt wurde und nie wieder gesperrt wird. Das führt dazu, dass alle anderen Threads die die Ressource benötigen, endlos auf deren Freigabe warten. Um das zu verhindern bietet die Standart Template Library (STL) eine „*lock_guard*“ Klasse die bei ihrer Zerstörung das Mutex selbständig wieder frei gibt. Diese Vorgehensweise seht man in Quellcodeauszug 1, wenn der „*lock_guard*“ in Zeile 354 nicht mehr im Scope ist, wird automatisch „*unlock()*“ aufgerufen. Bei jeder neu-

```
348 void WebsocketServer::addSession(std::shared_ptr<ws_session>
    ↪  ws_session_p)
349 {
350     ws_sessionsMutex_m.lock();
351     std::lock_guard<std::mutex> lg(ws_sessionsMutex_m,
        ↪  std::adopt_lock);
352     std::weak_ptr<ws_session> wp = ws_session_p;
353     ws_sessions_m.push_back(wp);
354 }
```

Quellcodeauszug 1: Methode „*addSession*“ der Websocket Server Klasse

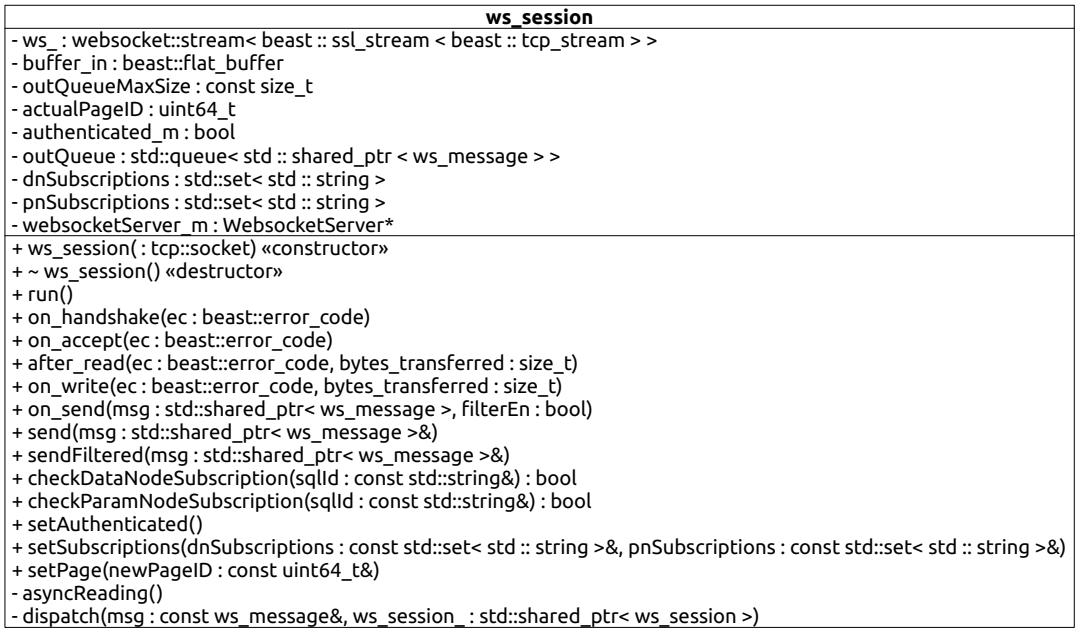


Abbildung 4.3: Klassediagramm der Klasse „*ws_session*“

en Websocket Verbindung wird ein neues „*ws_session*“ Objekt (Klassendiagramm Abbildung 4.3) instanziert und bekommt dabei einen Pointer auf das „*WebsocketServer*“ Objekt. Dies ermöglicht dem „*ws_session*“ Objekt, sich bei dem

„*WebsocketServer*“ Objekt anzumelden. Zusätzlich speichert das „*ws_session*“ Objekt den Pointer unter dem Attribut „*websocketServer_m*“ ab. Dies ermöglicht es beim Empfang eines kodierten Strings durch das „*ws_session*“ Objekt, ein „*ws_message*“ Objekt aus dem String zu konstruieren und die virtuelle Methode „*ws_dispatch*“ der „*WebsocketServer*“ Klasse aufzurufen, welche in der „*Backend*“ Klasse reimplementiert wird. Dort wird das „*ws_message*“ Objekt dann entsprechend seines Events interpretiert und der entsprechende Handler (beschrieben in Abschnitt 3.2.3, implementiert in der Methode „*ws_dispatch*“) ausgeführt. Daten die in die andere Richtung fließen, werden entweder direkt aus der Dispatcher Methode des Backends in eine einzelne Session geschickt (Aufruf der Methode „*send*“ des „*ws_session*“ Objekts), oder über die Methode „*publishToAllSessions*“ der „*WebsocketServer*“ Klasse. Dabei ist eine Art Publish/Subscribe Mechanismus implementiert bei dem Änderungen die DataNodes oder ParamNodes betreffen nur dann über die jeweilige Websocket Session an das Frontend gesendet werden, wenn eine entsprechende Seite dargestellt ist, welche die Nodes auch benutzt. Bei der Kommunikation mit dem Frontend werden, entsprechend

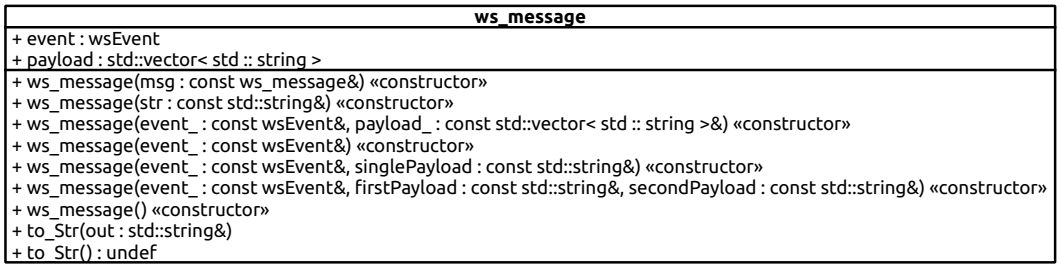


Abbildung 4.4: Klassediagramm der Klasse „*ws_message*“

des Abschnitts 3.2.3, Strings ausgetauscht. Diese Strings werden beim Empfang in einem „*ws_message*“ Objekt (Klassendiagramm in Abbildung 4.4) gespeichert und zum Dispatcher transportiert. Dort kann dann auf die einzelnen Komponenten der Nachricht (Event und Payloadvektor) einzeln zugegriffen werden. Außerdem ermöglicht die Klasse durch die Methode „*to_Str*“ es ihren Inhalt wieder als String, zum Transport an das Frontend, zu kodieren. Um den Code bei der Verwendung des „*ws_message*“ Objekts so kurz wie möglich zu halten, stehen zum Konstruieren eine Vielzahl von Konstruktoren zur Verfügung.

4.1.2 SqlClient

Die „*SqlClient*“ Klasse verwaltet die Verbindung zum SQL Server.
Auf dem SQL Server sind alle Daten gespeichert, die persistiert werden müssen. Die Datenstruktur des Servers ist in Abschnitt 3.3.1 beschrieben und in Abbildung 3.10 dargestellt. Diese Struktur ist auf dem SQL Server vorhanden und wird durch das Script in Quellcodeauszug 7 konstruiert. Die „*SqlClient*“ Klasse kann ein SQL Skript ausführen, indem sie es in einzelne Querys zerteilt und diese dann einzeln an den Server sendet. Das zerteilen eines Skript und das vorbereiten zur Ausführung, ist in der Methode „*prepareScript*“ implementiert. Beim testen der Methode hat sich herausgestellt, dass diese Funktionalität der „*sql*“ Client Klasse Skripte auszuführen, schwieriger zu implementieren ist als anfangs vermutet. Dies liegt daran, dass das Definieren von *stored Procedures* (siehe Abschnitt 2.3.3) es nötigt macht den

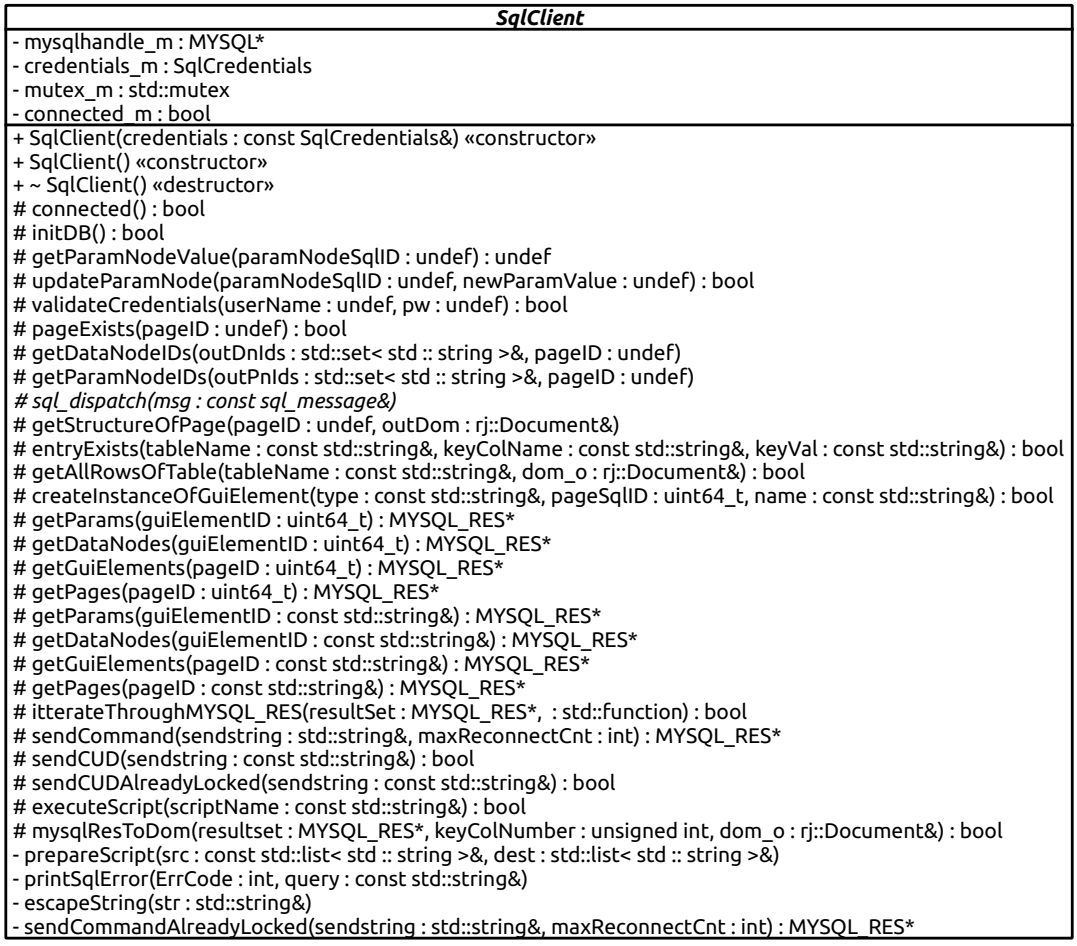


Abbildung 4.5: Klassediagramm der Klasse „*SqlClient*“

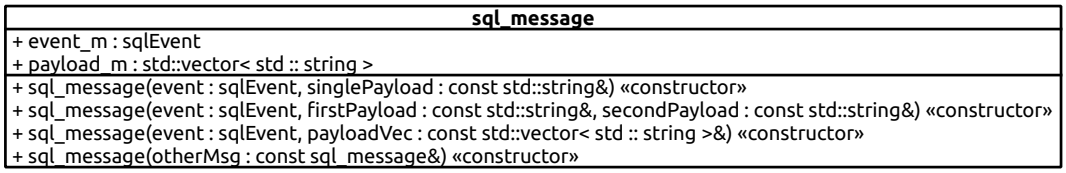


Abbildung 4.6: Klassediagramm der Klasse „*sql_message*“

4.1.3 OpcuaServer

Abbildung 4.7: Klassediagramm der Klasse „*OpcuaServer*“

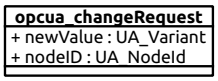


Abbildung 4.8: Klassediagramm der Klasse „*opcua_changeRequest*“

4.2 Frontend

```
{
  "socket": {
    "isConnected": true,
    "message": "",
    "reconnectError": false
  },
  "pageID": 1,
  "authenticated": true,
  "pageStruct": null
}
```

Quellcodeauszug 2: Datenstruktur Frontend

sieht man in Quellcodeauszug 6

```
{
  "pageID": 1,
  "parentId": null,
  "title": "ROOT_PAGE",
  "description": "the landing Page of the Visu",
  "subPages": [{
      "id": 2,
      "title": "SUB_PAGE",
      "description": "first sub page"
    },
    {
      "id": 3,
      "title": "SUB_PAGE2",
      "description": "second sub page"
    }
  ],
  "guiElements": {}
}
```

Quellcodeauszug 3: Datenstruktur Frontend - pageStruct

```
{
  "1": {
    "id": 1,
    "type": "button",
    "name": "der erste Button",
    "description": "just a stupid Button",
    "dataNodes": {},
    "paramNodes": {}
  }
}
```

Quellcodeauszug 4: Datenstruktur Frontend - guiElements

```
{
  "buttonState": {
    "id": 1,
    "name": "buttonState",
    "type": "Bool",
    "value": false,
    "description": "DataNode which holds the value of the Button a
    ↪  Button"
  },
  "colorSelector": {
    "id": 2,
    "name": "colorSelector",
    "type": "UInt8",
    "value": 0,
    "description": "DataNode to mux different colors to display a
    ↪  state"
  },
  "text": {
    "id": 3,
    "name": "text",
    "type": "String",
    "value": "ButtonText",
    "description": "text which is be displayed on the Button"
  }
}
```

Quellcodeauszug 5: Datenstruktur Frontend - dataNodes

```
{
  "buttonState": {
    "id": 1,
    "name": "buttonState",
    "type": "Bool",
    "value": false,
    "description": "DataNode which holds the value of the Button a
    ↪  Button"
  },
  "colorSelector": {
    "id": 2,
    "name": "colorSelector",
    "type": "UInt8",
    "value": 0,
    "description": "DataNode to mux different colors to display a
    ↪  state"
  },
  "text": {
    "id": 3,
    "name": "text",
    "type": "String",
    "value": "ButtonText",
    "description": "text which is be displayed on the Button"
  }
}
```

Quellcodeauszug 6: Datenstruktur Frontend - paramNodes

5 Zusammenfassung und Ausblick

Literaturverzeichnis

[1] Luber, S. [2018]. Definition: Was ist OPC UA?
URL: *https://www.bigdata-insider.de/was-ist-opc-ua-a-698144/*

[2] Scholz, P. [2005]. *Echtzeit, Echtzeitsysteme, Echtzeitbetriebssysteme*, Springer Berlin Heidelberg, Berlin, Heidelberg, chapter 3, S. 39–73.
URL: *https://doi.org/10.1007/3-540-27522-3_3*

[3] Schubert, M. [2007]. *„Das wird teuer“ — der EDV-Spezialist tritt auf*, Teubner, Wiesbaden, chapter 3, S. 40–49.
URL: *https://doi.org/10.1007/978-3-8351-9108-2_3*

[4] Studer, T. [2016]. *Das Relationenmodell*, Springer Berlin Heidelberg, Berlin, Heidelberg, chapter 2, S. 9–21.
URL: *https://doi.org/10.1007/978-3-662-46571-4_2*

Anhang

Quellcodeauszug 7: SQL Skript um die Datenbank zu initialisieren

```
USE WebVisu;
-- Pages
CREATE TABLE IF NOT EXISTS `Pages` (
  `ID` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `parentID` bigint(20) unsigned DEFAULT NULL,
  `title` varchar(30) NOT NULL,
  `description` varchar(250) NOT NULL,
  PRIMARY KEY (`ID`),
  KEY `Pages_Pages_FK` (`ParentID`),
  CONSTRAINT `Pages_Pages_FK` FOREIGN KEY (`parentID`) REFERENCES
  ↪  `Pages` (`ID`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
-- GuiElementTypes
CREATE TABLE `GuiElementTypes` (
  `ID` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `type` varchar(32) NOT NULL,
  `description` varchar(250) NOT NULL,
  PRIMARY KEY (`ID`),
  UNIQUE KEY `GuiElementTypes_UN` (`type`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
-- GuiElement
CREATE TABLE IF NOT EXISTS `GuiElements` (
  `ID` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `pageID` bigint(20) unsigned DEFAULT NULL,
  `typeID` int(10) unsigned NOT NULL,
  `name` varchar(30) NOT NULL,
  PRIMARY KEY (`ID`),
  KEY `GuiElement_Pages_FK` (`pageID`),
  KEY `GuiElement_GuiElementTypes_FK` (`typeID`),
  CONSTRAINT `GuiElement_GuiElementTypes_FK` FOREIGN KEY
  ↪  (`typeID`) REFERENCES `GuiElementTypes` (`ID`),
  CONSTRAINT `GuiElement_Pages_FK` FOREIGN KEY (`pageID`)
  ↪  REFERENCES `Pages` (`ID`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

```
32 -- DataTypes
33 CREATE TABLE `DataTypes` (
34   `type` varchar(100) NOT NULL,
35   PRIMARY KEY (`type`)
36 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
37 -- GuiElementDataNodeTemplates
38 CREATE TABLE `GuiElementDataNodeTemplates` (
39   `ID` int(10) unsigned NOT NULL AUTO_INCREMENT,
40   `writePermission` tinyint(1) DEFAULT false,
41   `type` varchar(100) NOT NULL,
42   `qualifiedName` varchar(100) DEFAULT 'unnamed',
43   `defaultValue` varchar(100) DEFAULT NULL,
44   `description` varchar(250) DEFAULT NULL,
45   PRIMARY KEY (`ID`),
46   KEY `GuiElementDataNodeTemplates_FK` (`type`),
47   CONSTRAINT `GuiElementDataNodeTemplates_FK` FOREIGN KEY (`type`)
     ↪ REFERENCES `DataTypes` (`type`)
48 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
49 -- GuiElementTypesDataNodesRel
50 CREATE TABLE `GuiElementTypesDataNodesRel` (
51   `guiElementTypeID` int(10) unsigned NOT NULL,
52   `dataNodeTemplateID` int(10) unsigned NOT NULL,
53   PRIMARY KEY (`guiElementTypeID`,`dataNodeTemplateID`),
54   KEY `GuiElementTypesDataNodesRel_FK_1` (`dataNodeTemplateID`),
55   CONSTRAINT `GuiElementTypesDataNodesRel_FK` FOREIGN KEY
     ↪ (`guiElementTypeID`) REFERENCES `GuiElementTypes` (`ID`) ON
     ↪ DELETE CASCADE,
56   CONSTRAINT `GuiElementTypesDataNodesRel_FK_1` FOREIGN KEY
     ↪ (`dataNodeTemplateID`) REFERENCES
     ↪ `GuiElementDataNodeTemplates` (`ID`) ON DELETE CASCADE
57 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
58 -- GuiElementDataNodes
59 CREATE TABLE `GuiElementDataNodes` (
60   `ID` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
61   `guiElementID` bigint(20) unsigned NOT NULL,
62   `typeID` int(10) unsigned NOT NULL,
63   `initValue` varchar(100) NOT NULL,
64   PRIMARY KEY (`ID`),
65   KEY `GuiElementDataNodes_GuiElement_FK` (`guiElementID`),
66   KEY `GuiElementDataNodes_FK` (`typeID`),
67   CONSTRAINT `GuiElementDataNodes_FK` FOREIGN KEY (`typeID`)
     ↪ REFERENCES `GuiElementDataNodeTemplates` (`ID`) ON DELETE
     ↪ CASCADE,
```

```
68    CONSTRAINT `GuiElementDataNodes_GuiElement_FK` FOREIGN KEY
      ↪  (`guiElementID`) REFERENCES `GuiElements` (`ID`) ON DELETE
      ↪  CASCADE
69  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
70  -- GuiElementParamTemplates
71  CREATE TABLE `GuiElementParamTemplates` (
72    `ID` int(10) unsigned NOT NULL AUTO_INCREMENT,
73    `type` varchar(100) NOT NULL,
74    `qualifiedName` varchar(100) DEFAULT 'unnamed',
75    `defaultValue` varchar(100) DEFAULT NULL,
76    `description` varchar(250) DEFAULT NULL,
77    PRIMARY KEY (`ID`),
78    KEY `GuiElementParamTemplates_FK` (`type`),
79    CONSTRAINT `GuiElementParamTemplates_FK` FOREIGN KEY (`type`)
      ↪  REFERENCES `DataTypes` (`type`)
80  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
81  -- GuiElementTypesParamsRel
82  CREATE TABLE `GuiElementTypesParamsRel` (
83    `guiElementTypeID` int(10) unsigned NOT NULL,
84    `paramTemplateID` int(10) unsigned NOT NULL,
85    PRIMARY KEY (`GuiElementTypeID`,`ParamTemplateID`),
86    KEY `GuiElementTypesParamsRel_FK` (`paramTemplateID`),
87    CONSTRAINT `GuiElementTypesParamsRel_FK` FOREIGN KEY
      ↪  (`paramTemplateID`) REFERENCES `GuiElementParamTemplates`
      ↪  (`ID`) ON DELETE CASCADE,
88    CONSTRAINT `GuiElementTypesParamsRel_FK_1` FOREIGN KEY
      ↪  (`guiElementTypeID`) REFERENCES `GuiElementTypes` (`ID`) ON
      ↪  DELETE CASCADE
89  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
90  -- GuiElementParams
91  CREATE TABLE `GuiElementParams` (
92    `ID` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
93    `guiElementID` bigint(20) unsigned NOT NULL,
94    `typeID` int(10) unsigned NOT NULL,
95    `value` varchar(100) NOT NULL,
96    PRIMARY KEY (`ID`),
97    KEY `GuiElementParams_GuiElement_FK` (`guiElementID`),
98    KEY `GuiElementParams_FK` (`typeID`),
99    CONSTRAINT `GuiElementParams_FK` FOREIGN KEY (`typeID`)
      ↪  REFERENCES `GuiElementParamTemplates` (`ID`) ON DELETE
      ↪  CASCADE,
100   CONSTRAINT `GuiElementParams_GuiElement_FK` FOREIGN KEY
      ↪  (`guiElementID`) REFERENCES `GuiElements` (`ID`)
```

```
101 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
102 CREATE TABLE `Credentials` (
103   userName varchar(100) NOT NULL,
104   pwHash varchar(100) NOT NULL,
105   PRIMARY KEY (`userName`)
106 )  ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```