



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

|

# **Abbildungsverzeichnis**

# 1 Einführung

## 1.1 Motivation

Durch mein Studium der Elektrotechnik mit Vertiefung in die Automatisierungstechnik konnte ich Eindrücke in die Vorgehensweise und Möglichkeiten der **gui!** (**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 kompiliert 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 plattformgebunden 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 **scada!** (**scada!**) System mit Webfrontend. Durch ein **poc!** (**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 **http!** (**http!**) Verbindung. Der Client sendet eine **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 **tcp!** (**tcp!**) Verbindung auf die so lange bestehen bleibt bis einer der Teilnehmer die Verbindung aktiv beendet, oder Ein Teilnehmer nichtmehr erreichbar 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 auch eine Websocket Verbindung verschlüsselt werden dies erkennt man am Protokollpräfix der **url!** (**url!**). Bei einer unverschlüsselten Verbindung lautet der Präfix „*ws://*“ und bei einer verschlüsselten „*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

**dbms!** (**dbms!**) zusammen. Dabei stellt die Datenbasis den Speicher der Datenbank dar und das **dbms!**, das Programm durch das der Zugriff auf die Datenbasis geschieht. Das **dbms!** ist notwendig, um konkurrierende Zugriffe auf die Datenbasis, von mehreren Nutzern zu Verwalten. [? ] 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.

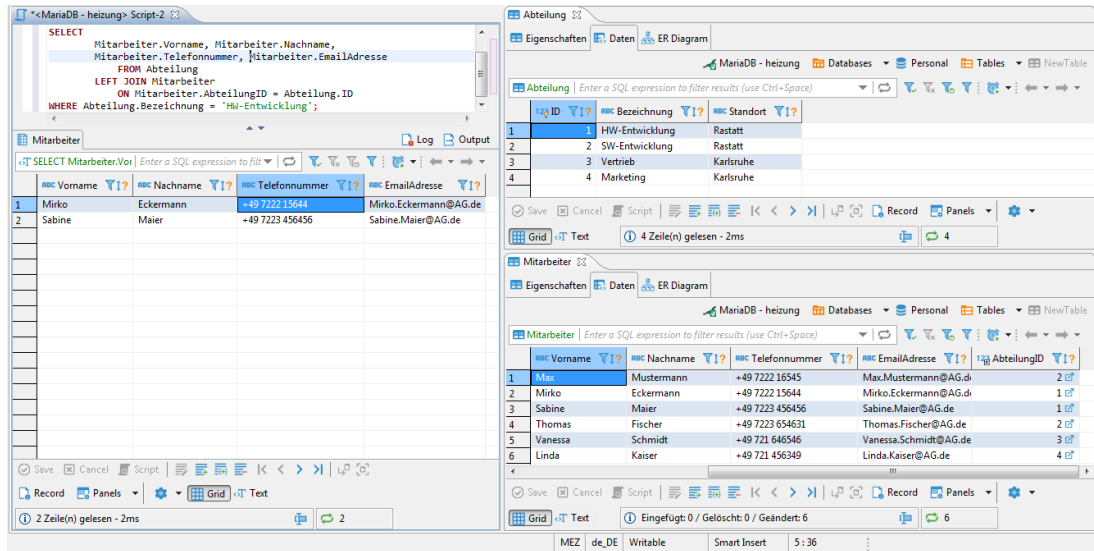
Jede Relation hat Attribute mit einem in der Relation einzigartigen Attributnamen. Die Domäne eines Attributs ist die Menge aller Werte die ein Attribut annehmen kann. Ist der Wert eines Attributs unbekannt oder noch nicht bestimmt, ist der Wert des Attributs „NULL“ [? ]. Werden Daten nun in einer Datenbank 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 den Relationen nun zur 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 (**crud!** (**crud!**) Operationen) zu machen. Die Schnittstelle unterstützt meist die Sprache **sql!** (**sql!**). Diese Sprache ist in Abschnitt ?? 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 verknüpft 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 heißt 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 eine 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 einer relationalen **dbms!** existiert. Er ermöglicht es, einen Verweis auf eine andere Tabelle zu definieren. Zur Veranschaulichung sei



folgendes Beispiel einer relationalen Datenstruktur, auf einem **sql!** Server gegeben. Wie in Abbildung ?? zu sehen, besteht die Datenbank *Personal* aus zwei



**Abbildung 2.1:** Beispiel Datenstruktur

Tabellen. Die erste Tabelle hat die Bezeichnung (*Abteilung*) mit den Spalten *ID*, *Bezeichnung* sowie *Standort*. Die Spalte *ID* ist als *primaryKey* deklariert. Die zweite Tabelle (*Mitarbeiter*) enthält die Spalten *Personalnummer*, *Vorname*, *Nachname*, *Telefonnummer*, *EmailAdresse*, *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 Fremdschlüssel definiert, der von der Spalte *AbteilungID* der Tabelle *Mitarbeiter* auf die Tabelle *Abteilung*, unter Benutzung deren Primärschlüssels, verweist. Die Benutzung eines solchen Fremdschlüssels ist nicht zwingend notwendig um die Verknüpfung nach dieser Regel zu ermöglichen, Es ermöglicht aber die selbstständige Erhaltung der Datenbank der referenziellen Integrität. Dabei hat man bei der Erstellung des Fremdschlüssels die Möglichkeit zu entscheiden, 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 auf den referenzierenden Datensatz löscht (*on delete cascade*). Bei beiden Verfahren bleibt die referenzielle Integrität erhalten, dass heist es gibt nach dem Löschen keine Referenzen die nicht aufgelöst werden können.

## 2.3.2 sql!

**sql!** ist eine Abkürzung für **sql!**. 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 ??, 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 entstanden Tabelle auszugeben, welche die Abteilungsbezeichnung *HW-Entwicklung* tragen. Die Query für diese Operation ist in Abbildung ?? abgebildet. Die Antwort des Servers ist in Abbildung ?? unten links zu sehen. Rechts

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

**Abbildung 2.2:** Beispiel sqlQuery - Select mit Join

seht man die Datensätze der beiden Quelltabellen des Queries. 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, man muss durch jedes Element iterieren. Die Datenbank hat zur Lösung dieses Problems bessere Algorithmen hinterlegt (Stichwort: binärer Suchbaum, Hash-Maps...).

### 2.3.3 Stored Procedures

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 session-bezogene globale Variablen als *OUT* Argument. Desweiteren haben Prozeduren

entgegen Funktionen die Möglichkeit, SQL-Queries zur Laufzeit zusammenzusetzen und auszuführen. Funktionen können, in **sql!** Queries benutzt werden, Prozeduren nicht. Prozeduren werden durch ein *CALL* Befehl aufgerufen.

## 2.4 OPC UA

**opcua!** (**opcua!**) ist ein Thema dass immer wieder mit Industrie 4.0 in Verbindung gebracht wird. ? ] fasst **opcua!** 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 **opcua!**, er definiert nicht nur wie Daten ausgetauscht werden, sondern auch wie die Daten zu interpretieren sind (Semantik). Dabei bedient sich ein **opcua!** Server einem eingebauten Informationsmodell. Das 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 **opcua!** Schnittstelle (Server). Dieser Server hat ein Pumpenobjekt mit der Bezeichnung „*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 *opcua* Server unterstützt aber nicht nur Primitive Datentypen, sondern auch Objekte wie das Pumpenobjekt „*pumpe\_xyz\_A*“ des Herstellers A. Nun möchte man diese Pumpe des Herstellers A durch ein **plc!** (**plc!**) des Herstellers B auslesen bzw steuern. Bei einer proprietären 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 dann Die Pumpe ansteuern könnte. Der Hersteller wäre also gezwungen, für jeden größeren **plc!** Hersteller, verschiedene Dokumente zur Verfügung zu stellen. **opcua!** geht an dieser Stelle einen anderen Weg. Die OPC Foundation stellt eine Menge an **opcua!** 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 Ihr Pumpenklasse (*pumpe\_xyz\_A*) von dieser Pumpenklasse ab und erweitert die Klasse um ihrer 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 der OPC Foundation 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 **opcua!** 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 **opcua!** das **ip!** (**ip!**). Damit ist eine horizontale sowie vertikale Vernetzung innerhalb der Automatisierungspyramide möglich. **opcua!** unterstützt 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.

## 2.5 Echtzeit

Laut [?] 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:

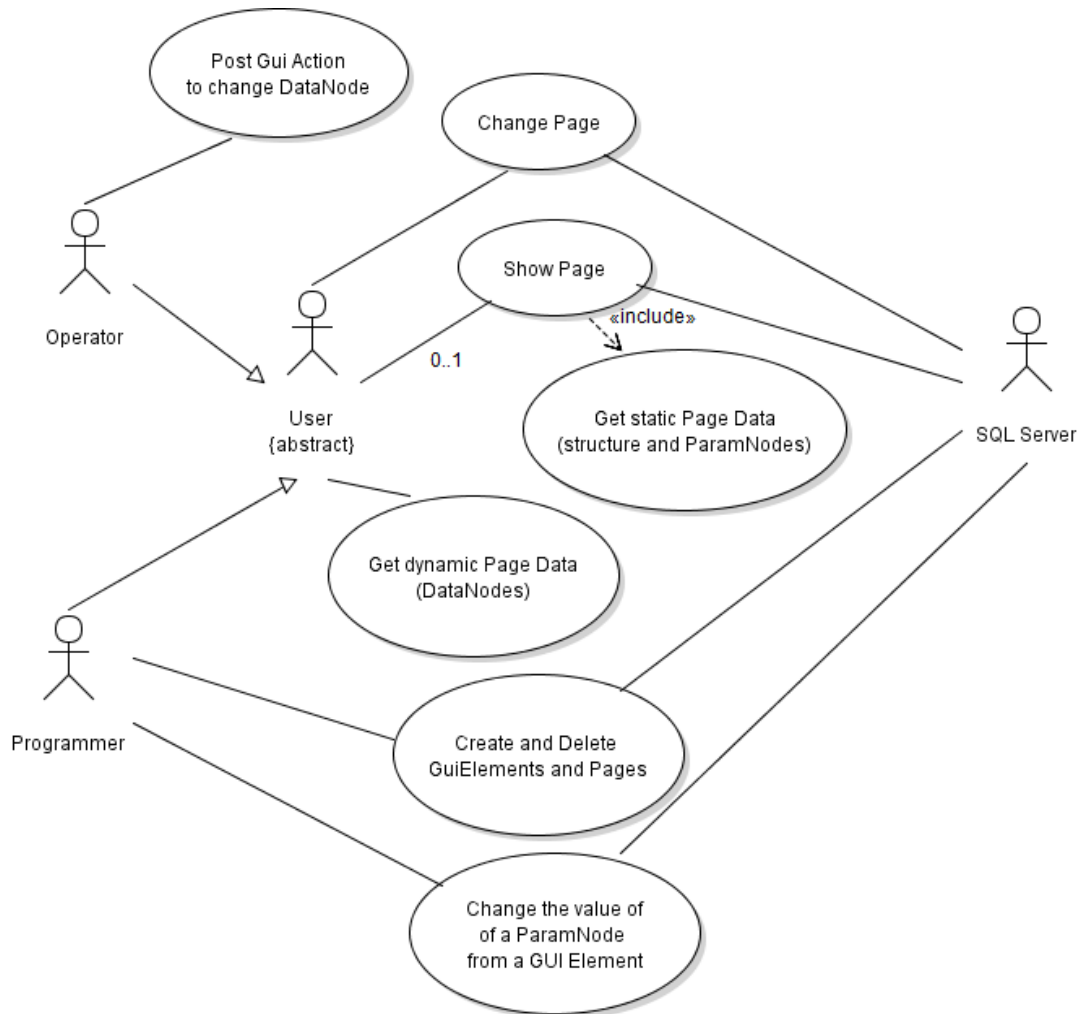
- Jede Komponente des Systems die an dem Prozess beteiligt ist, muss selbst echtzeitfähig sein.
- Die Rechenzeit der verwendeten Algorithmen muss endlich sein.
- Die verwendeten Transportprotokolle um dem Prozess Daten zuzuführen bzw. Daten zu entnehmen müssen die Echtzeitfähigkeit unterstützen.

Wenn die obere Definition nun auf das Ethernet oder das **tcp!** Protokoll angewandt 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 den BUS abgebrochen und nach einer zufälligen Zeit erneut versucht. Dieses Verfahren nennt man **CSMA/CD!** (**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. Man muss anmerken man immernoch nicht vollständig Echtzeitfähig Daten übertragen kann, denn das Netzwerk muss auch auf die aufkommende Datenlast ausgelegt sein. 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 Zeit definiert und endlich ist.

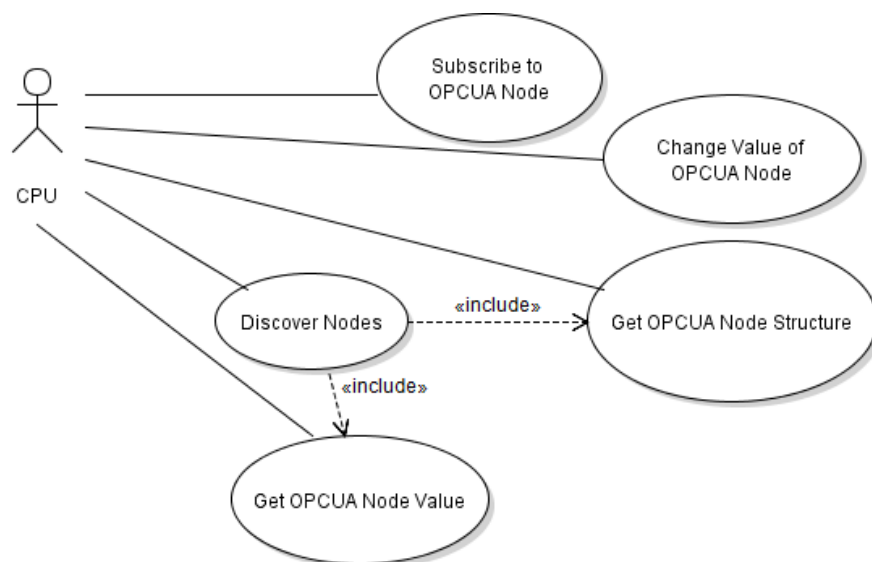
## **3 Systementwurf**

### 3.1 Use-Cases



**Abbildung 3.1:** Anwendungsfalldiagramm des **scada!** Systems I

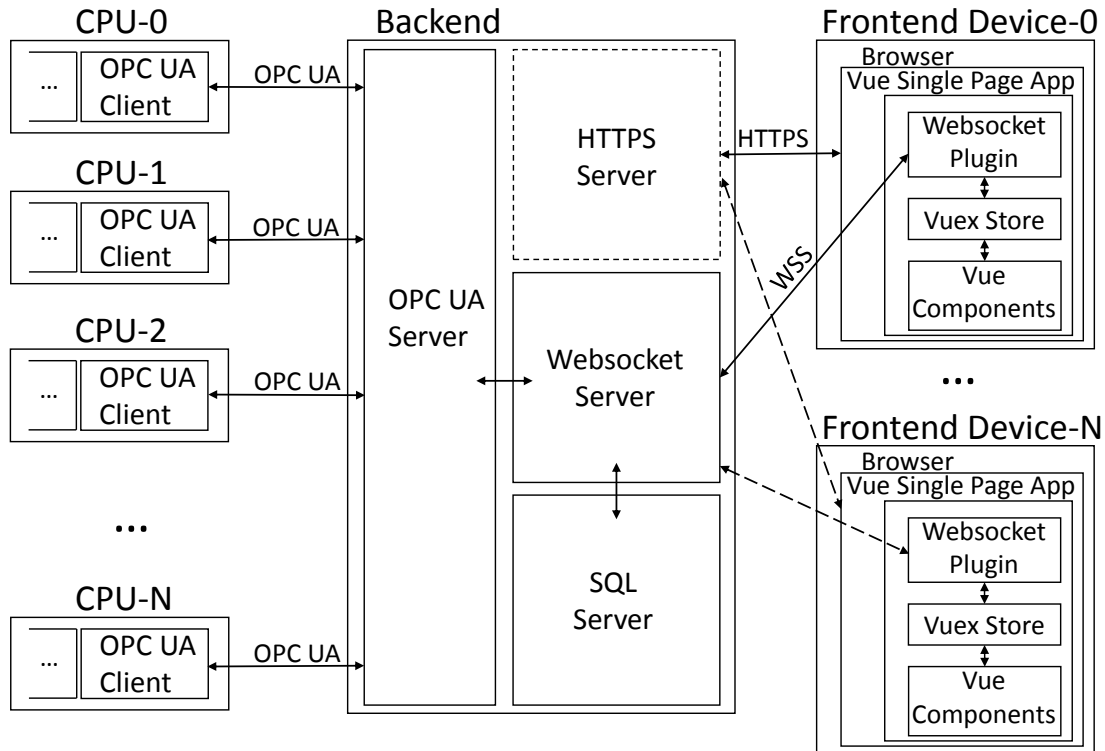
Aus der Zielsetzung der Arbeit ergeben sich die Anwendungsfalldiagramme in Abbildungen ?? und ?. Das Diagramm in Abbildung ?? beschreibt abstrakt die Anwendungsfälle, die das **scada!** System für den Anwender, in der Rolle des Bedieners (Operator), sowie des Programmierers (Programmer) erfüllen muss. Analog dazu beschreibt das Anwendungsfalldiagramm in Abbildung ?? 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:** Use-Case Diagramm des **scada! Systems II**



## 3.2 Architektur



**Abbildung 3.3:** Kommunikationsmodell des **scada!** Systems.

Die CPUs sind per **opcua!** an das Scada System angebunden und haben die Rolle eines **opcua!** Clients. Die Webapplikation im Browser kommuniziert über **wss!** sowie **https!** verschlüsselt mit dem Backend.

Wie in Abbildung ?? dargestellt, lässt sich die Architektur des **scada!** Systems in **plc!**s, Backend und Frontends unterteilen. Unter Frontend (Abschnitt ??) versteht man die (in diesem Fall) grafische Schnittstelle, die es dem Nutzer ermöglicht mit dem System zu interagieren. Das Backend (Abschnitt ??) fasst den Rest des Systems zusammen.

### 3.2.1 Backend

Das Backend besteht aus den folgenden vier Komponenten:

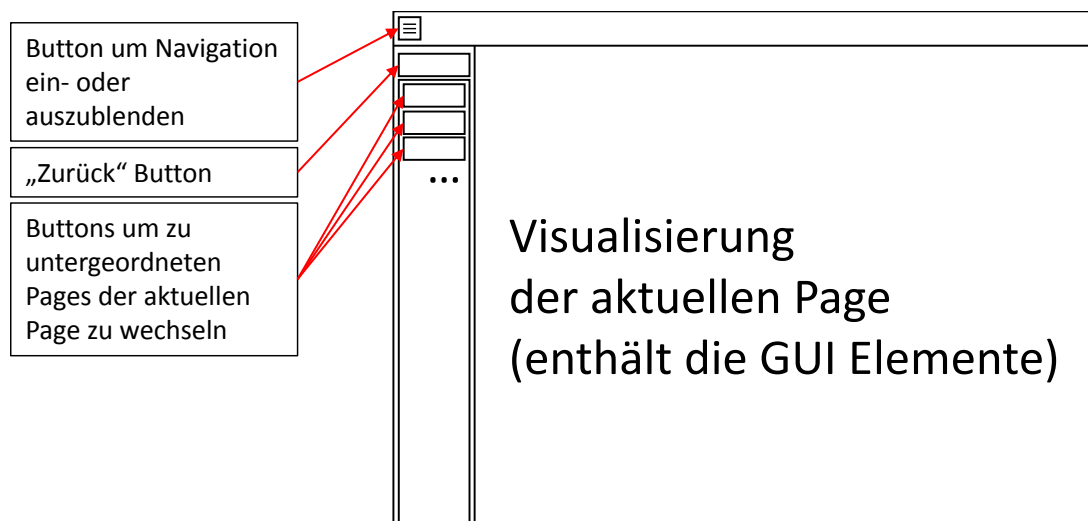
- Einem **https!** (**https!**) Server
- Einem **wss!** (**wss!**) Server
- Einem **sql!** Server

- Einem **opcua!** Server

Damit ergeben sich, wenn man das Backend von außen betrachtet, als Schnittstellen **opcua!**, **wss!** und **https!**. Über diese Schnittstellen stellt das Backend den **plc!**s, 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 **html!** (**html!**), die **css!** (**css!**), sowie die **js!** (**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 **opcua!** Server hält die variablen Daten des Systems und bietet diese den angeschlossenen **plc!**s 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

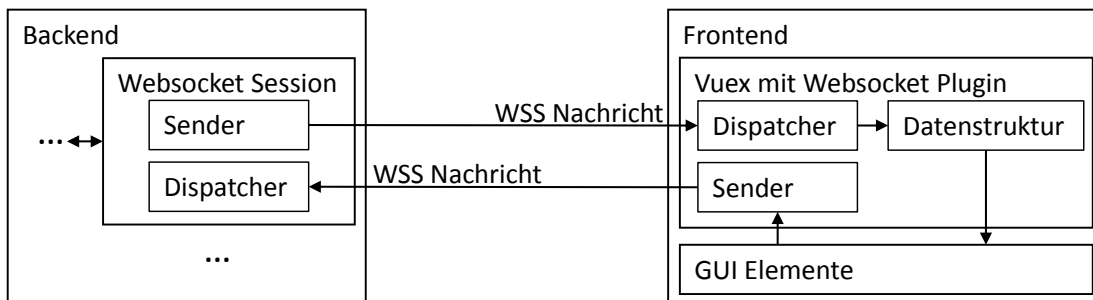
Das Frontend ist eine Webapplikation, mit Vue.js als Webframework. Beim Laden der Seite wird eine **wss!** Verbindung zum Backend aufgebaut. Ohne Authentifizierung zeigt das Frontend eine Seite an, die es dem Nutzer ermöglicht sich durch individuelle Zugangsdaten (Benutzername und Passwort) zu authentifizieren. Werden auf dieser Seite gültige Zugangsdaten eingegeben, ist die **wss!** Session authentifiziert und zeigt das Layout in Abbildung ?? an. Das Layout bein-



**Abbildung 3.4:** Layout des Frontends bei einer authentifizierten **wss!** Session

haltet oben links einen Button um die Navigation der Seite ein und auszublen-

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 ??), welche die aktuell angezeigte Seite betreffen. Sie ist in Abschnitt ?? 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 ?? beschrieben. Innerhalb der Webapplikation existiert eine Page Komponente die, entsprechend der **gui!** Elemente in der globalen Datenstruktur, **gui!** Element Komponenten (*guiElement*) instanziiert 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 ?? 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



**Abbildung 3.5:** Schema des Zugriffs auf die Datenstruktur des Frontends

der Dispatcher die Struktur des Frontends ändert ist in Abschnitt ?? 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

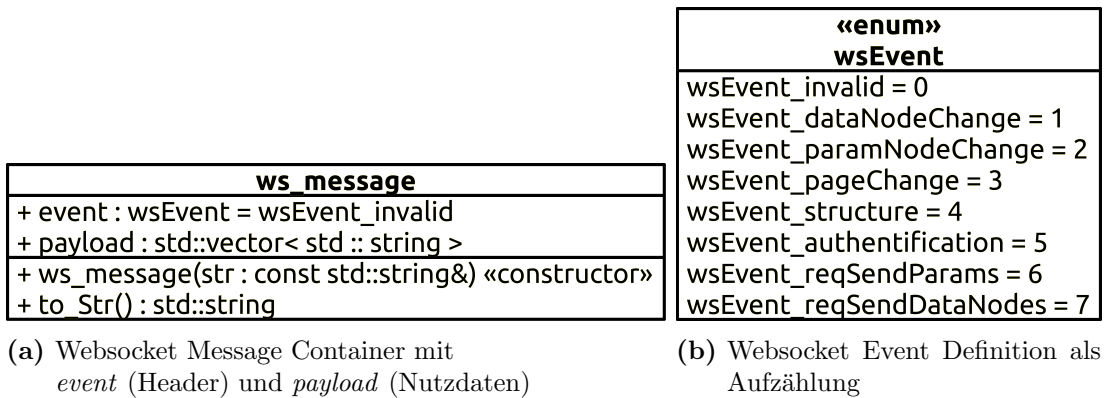
- ... so einfach wie möglich sein.
- ... alle Datentypen unterstützen, die das Backend definiert.
- ... weitestgehend symmetrisch sein (Die Nachrichten vom Server zum Client haben eine ähnliche Gestalt wie die vom Client zum Server).
- ... möglichst leichtgewichtig sein.
- ... möglichst frei von Zuständen 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 ?? 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. Diese Aufzählung (*wsEvent*) ist in Abbildung ?? 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

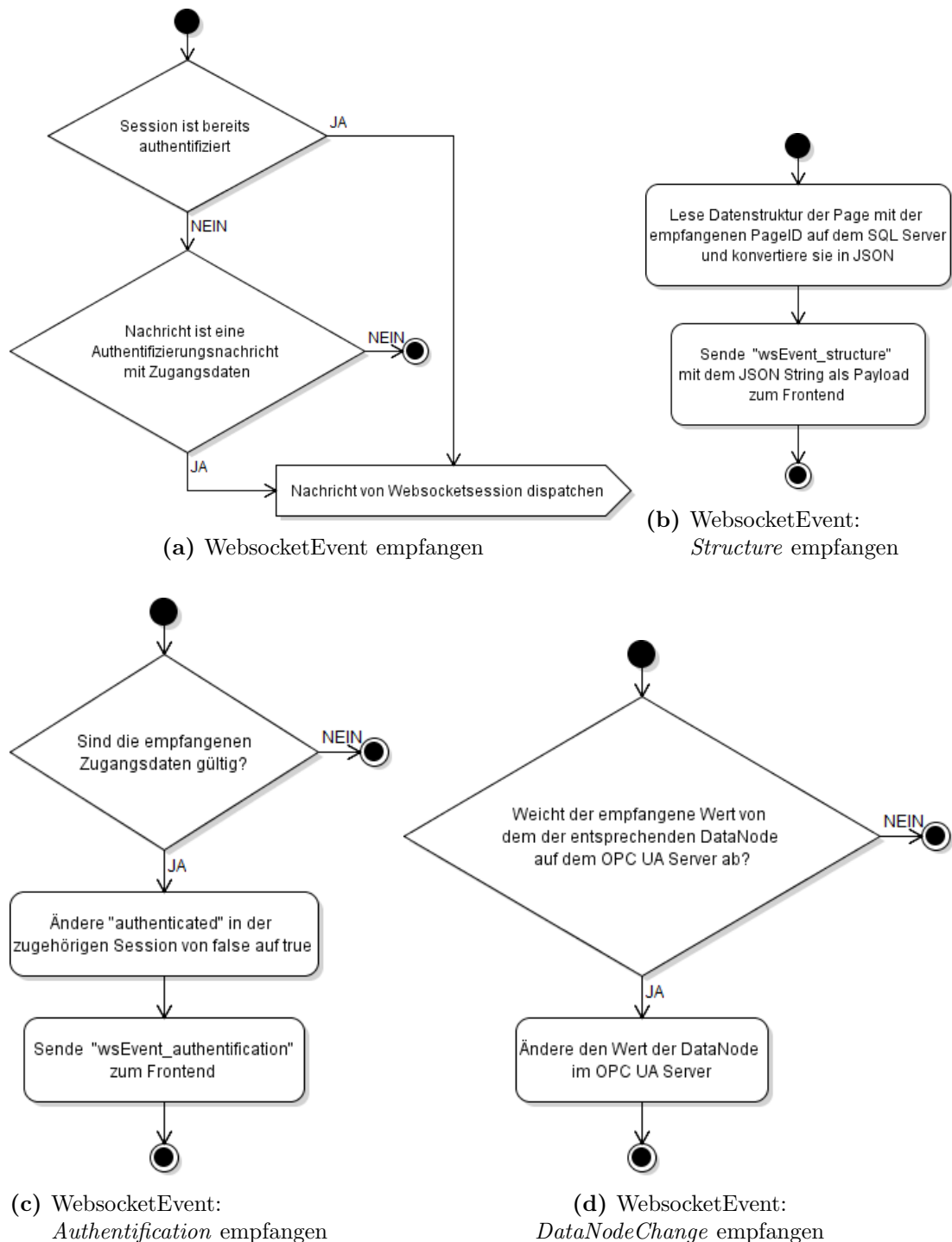


**Abbildung 3.6:** WebSocket Message Container und Event

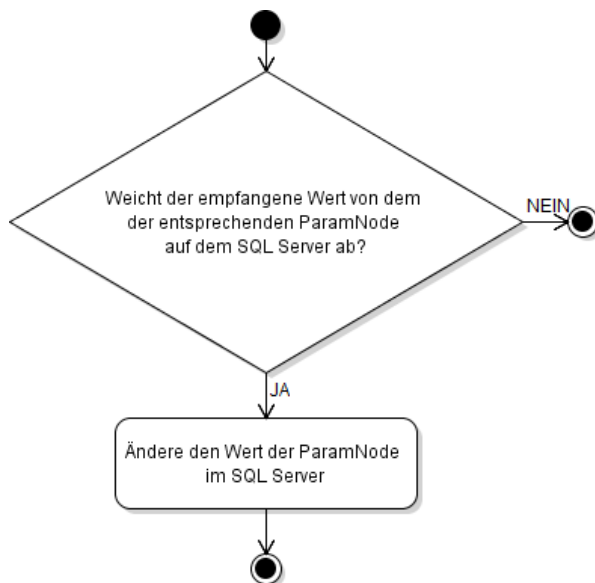
und durch ein Semikolon getrennt zu einem String zusammenzusetzen. Dabei stellt das erste Feld in einem versendeten String immer das kodierte Event dar. Die restlichen Felder entsprechen dem Nutzdatenvektor der *ws\_message*. Diese Bildungsvorschrift ist umkehrbar, solange der Nutzdatenvektor kein String 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 ?? dargestellten Events. Das erste Event (*wsEvent\_invalid*) ist der Standardwert 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 Dispatcher 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 ?? 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 ??, sowie durch die Diagramme in Abbildung ?? definiert.

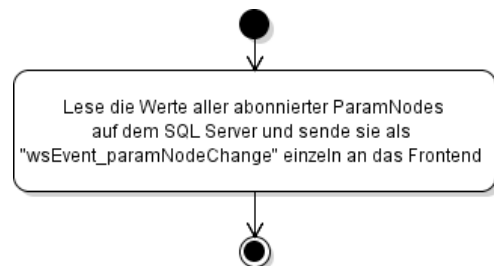
Das Frontend bedarf keines solchen Authentifizierungsmechanismus, da dies bereits im **wss!** Protokoll implementiert ist. Wie auch bei **https!** geschieht das durch ein Serverzertifikat des Backends, dass durch eine externe **ca!** (**ca!**) beim Verbindungsaufbau verifiziert werden muss. Die Handler des Frontends sind in Abbildung ?? dargestellt.



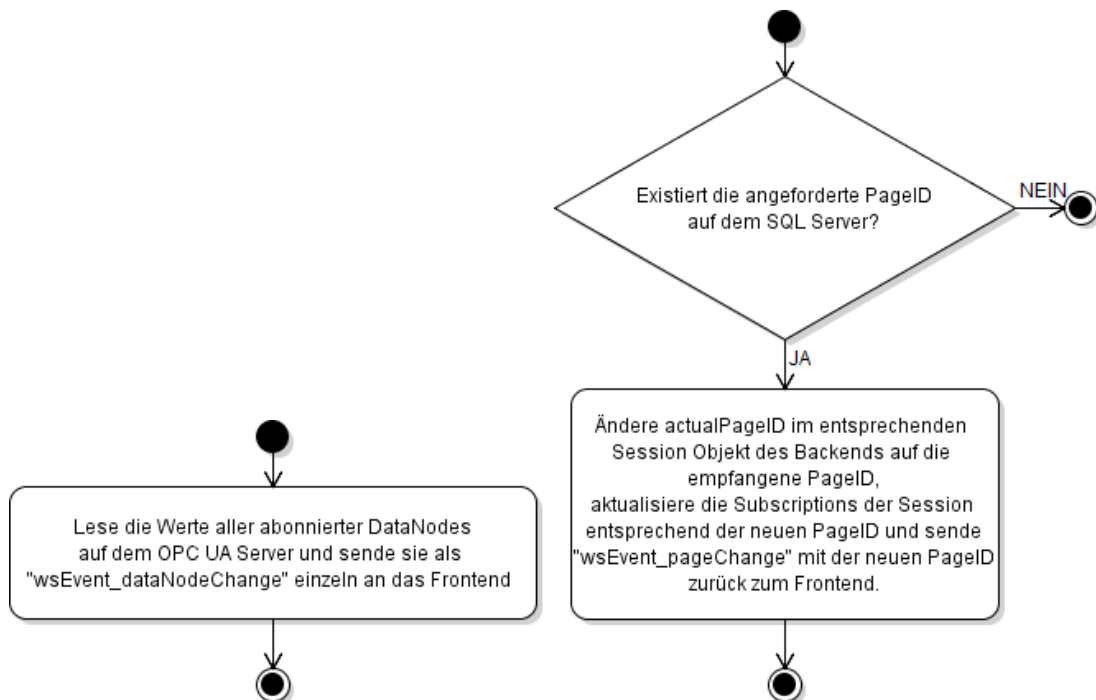
**Abbildung 3.7:** Aktivitätsdiagramme des Dispatchers im Backend I



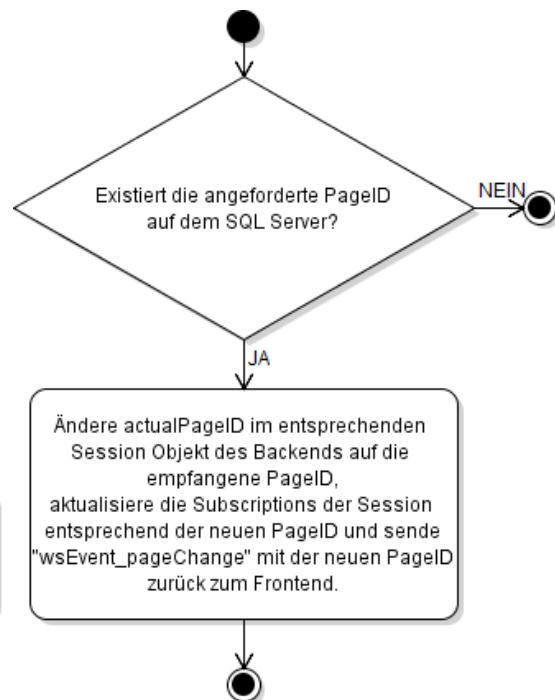
(a) WebsocketEvent:  
*ParamNodeChange* empfangen



(b) WebsocketEvent:  
*ReqSendParamNodes* empfangen

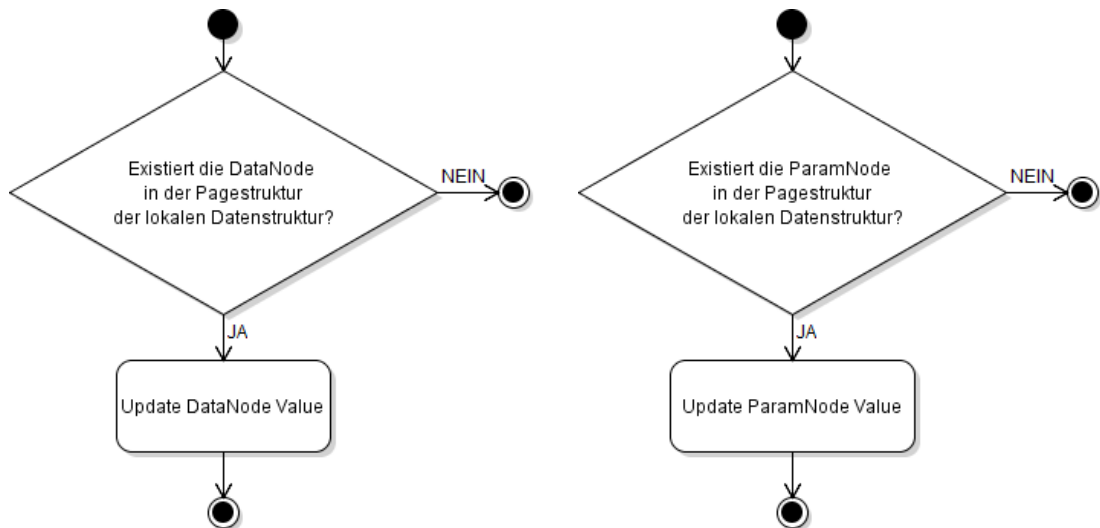


(c) WebsocketEvent:  
*ReqSendDataNodes* empfangen



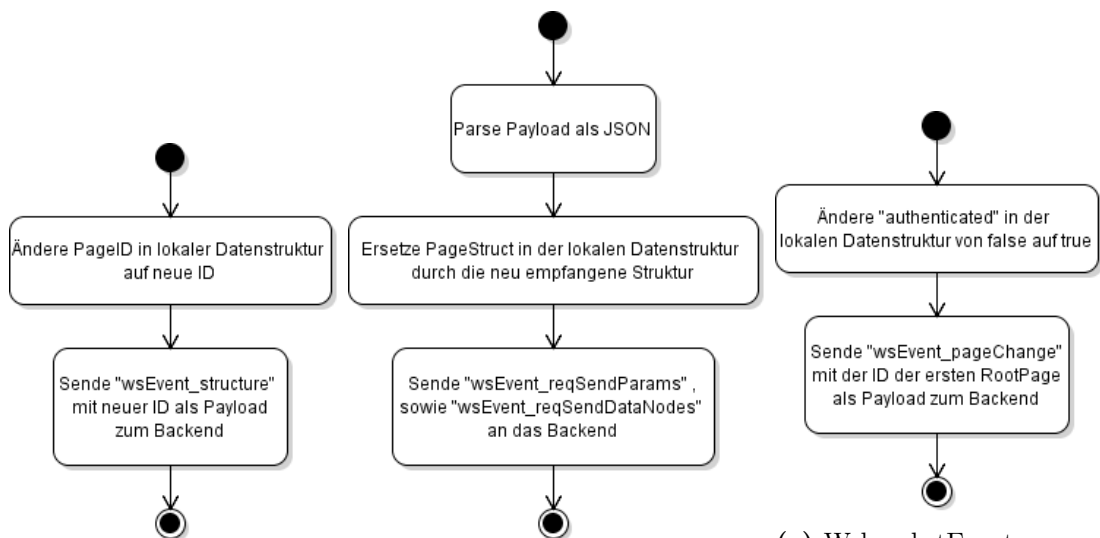
(d) WebsocketEvent:  
*PageChange* empfangen

**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:  
*Authentication* empfangen

**Abbildung 3.9:** Aktivitätsdiagramme des Dispatchers im Frontend



## 3.3 Datenstruktur

### 3.3.1 Backend

Das abgebildete **erd!** (**erd!**) in Abbildung ??, 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. Ist der Wert dieses Fremdschlüssels *NULL* so hat die jeweilige Page keine übergeordnete Page und liegt damit im Wurzelverzeichnis.

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, erreichbar. 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 ??).

Der **opcua!** Server hält alle DataNodes der **gui!** Elemente. Diese **gui!** Elemente werden entsprechen der Pages, denen sie zugeordnet sind, auf dem **opcua!** Server abgelegt und sind den **plc!**s dort zugänglich. Ob eine DataNode von den **plc!**s 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 **opcua!** Server, wird dieses Flag ausgelesen und direkt in die Konfiguration des **opcua!** Servers übernommen. Der native Daten-

typ, mit dem eine DataNode auf dem **opcua!** 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 **opcua!** 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 **opcua!** Server den Typ BaseObjectType. Auf einen Spezialisierung dieser Typdefinition für Pages und **gui!** Elements auf dem **opcua!** Server wird absichtlich verzichtet, da sie nur zur Strukturierung der DataNodes, auf dem **opcua!** Server, genutzt werden.

### 3.3.2 Frontend

Die Datenstruktur in der Webapplikation wird in dem Vue Plugin Vuex gespeichert. Die Struktur ist in Abbildung ?? dargestellt

Sie ist eine Kopie einer einzelnen Page als **json!** (**json!**)

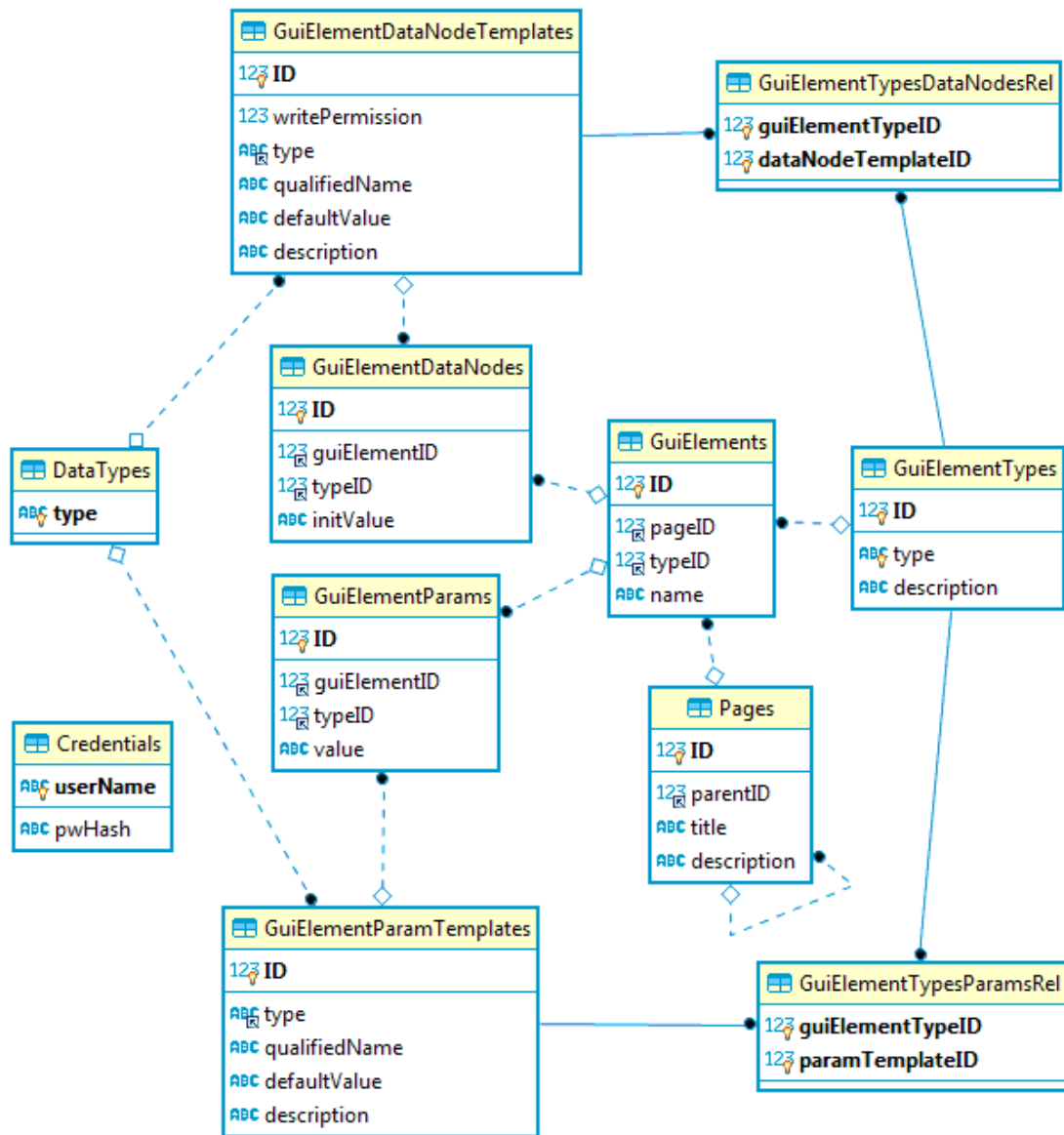
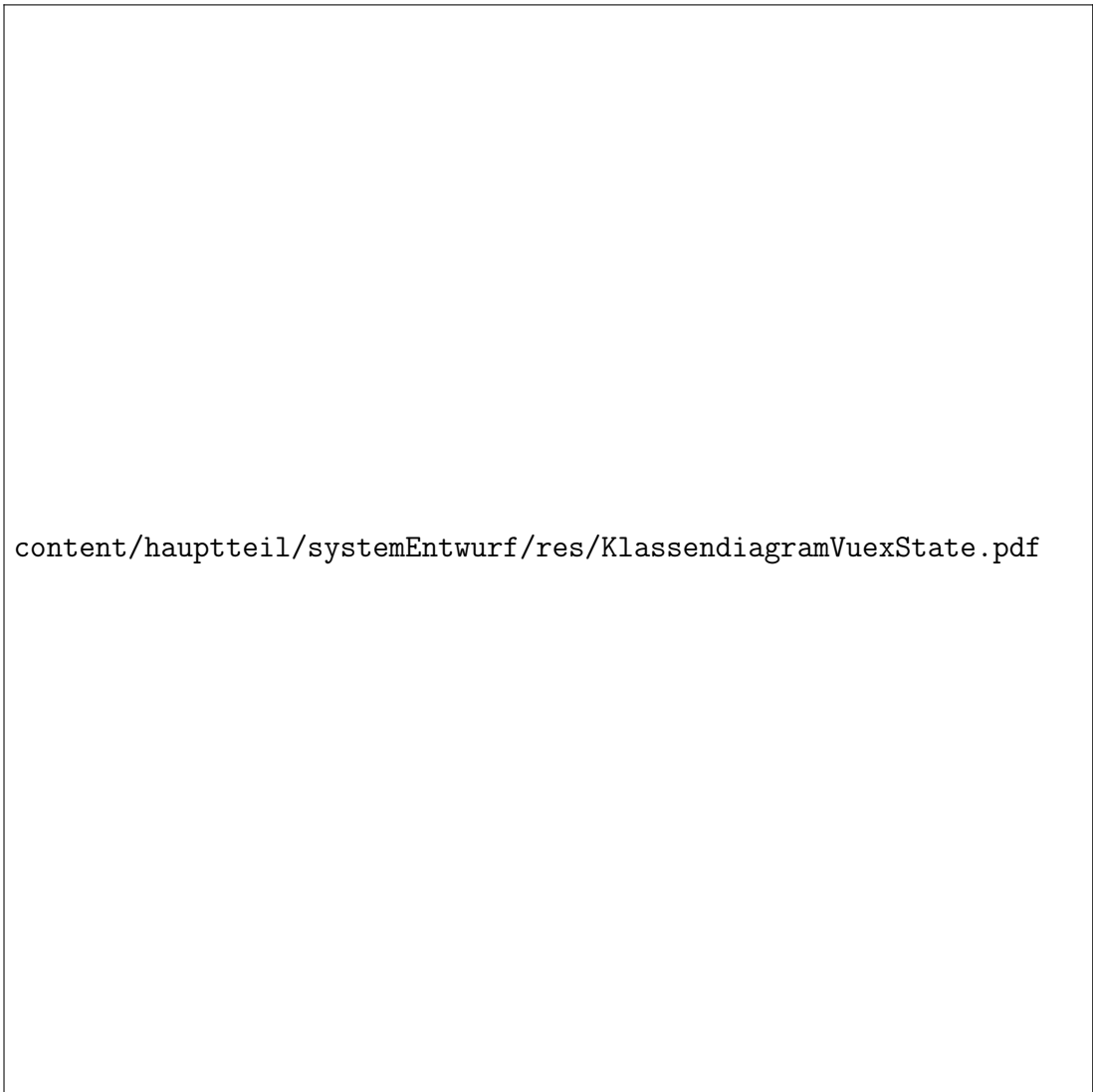


Abbildung 3.10: erd! des scada! Systems.

Darstellung zeigt die einzelnen Tabellen der Datenbank auf dem sql! Server und deren Beziehungen zueinander.



`content/hauptteil/systemEntwurf/res/KlassendiagramVuexState.pdf`

**Abbildung 3.11:** Schema des Zugriffs auf die Datenstruktur des Frontends

## 4 Umsetzung des Proof of Concept

Ein **poc!** ist eine Möglichkeit um die Realisierbarkeit des Systementwurfs aus Kapitel ?? zu belegen. In diesem **poc!** wurde sich auf die wesentliche Funktionalität eines **scada!** Systems beschränkt. Dabei wird Anhand der Architektur in Abschnitt ?? eine Webapplikation als Frontend und eine ausführbare Anwendung als Backend programmiert und anschließend getestet. Die Dokumentation dieses PoC ist den folgenden zwei Kapiteln (??) enthalten.

## **4.1 backend**

??

### **4.1.1 HIER KLASSEN**

## 4.2 Frontend

## **5 Zusammenfassung und Ausblick**