Bachelorarbeit

Untersuchung von Optimierungsmöglichkeiten für Zugriffe auf ein OPC UA Modell aus einer clientseitigen Web-Anwendung

Zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
Angewandte Informatik
im Fachbereich Ingenieur- und Naturwissenschaften
der Hochschule Merseburg

Autor:
Rim Chaari
Matrikel-Nr. 24495

Erstprüfer:
Prof. Dr. Thomas Meier
(Hochschule Merseburg)

Zweitprüfer:
Nico Scheithauer (M.Eng.)
(Hochschule Merseburg)

6. März 2025

Abstract

This bachelor thesis deals with the optimization of access to the OPC UA data model by developing an API based on a single session and recursive data processing. In addition, a frontend was developed that visualizes the data and enables efficient interaction with the OPC UA model.

The starting point was a system consisting of REST API together with a frontend that had significant performance issues due to its architecture.

The use of a single session to access all nodes reduced the number of connections and shortened latency. To make this possible, recursive data processing was used. This simplifies navigation through the hierarchical structure of the data model and optimizes queries.

The choice of GraphQL offers advantages such as targeted data queries and support for subscriptions for real-time updates. This allows users to track changes in the nodes directly in the frontend without having to send repeated queries to the server. In addition, a caching mechanism has been implemented in the frontend, which further reduces response times by caching frequently retrieved data.

The evaluation shows that the above-mentioned improvements significantly increase performance and user-friendliness. The combination of a single session, recursive data processing and GraphQL in the backend on the one hand and caching in the frontend on the other represents a more powerful solution than the existing REST-API for accessing complex Data models and contributes to the optimization of data access architectures in modern applications.

Abstrakt

Diese Bachelorarbeit befasst sich mit der Optimierung des Zugriffs auf das OPC UA-Datenmodell durch die Entwicklung einer API, die auf einer einzigen Sitzung und Rekursive Datenverarbeitung basiert. Zusätzlich wurde ein Frontend entwickelt, das die Daten visualisiert und eine effiziente Interaktion mit dem OPC UA-Modell ermöglicht.
Ausgangspunkt war ein System bestehend aus REST-API zusammen mit einem Frontend, die erhebliche Performance-Probleme aufgrund ihrer Architektur aufwies.
Die Nutzung einer einzigen Sitzung zum Zugriff auf alle Knoten hat die Anzahl der Verbindungen reduziert und die Latenzzeit verkürzt. Um dies zu ermöglichen, wurde rekursive Datenverarbeitung verwendet. Dies vereinfacht die Navigation durch die hierarchische Struktur des Datenmodells und optimiert die Abfragen.
Die Wahl von GraphQL bietet Vorteile wie gezielte Datenabfragen und die Unterstützung von Subscriptions für Echtzeit-Updates. Damit können Benutzer Änderungen in den Knoten direkt im Frontend verfolgen, ohne wiederholte Abfragen an den Server zu senden. Darüber hinaus wurde ein Caching-Mechanismus im Frontend implementiert, der die Antwortzeiten weiter reduziert, indem häufig abgerufene Daten zwischengespeichert werden.
Die Evaluation zeigt, dass die oben erwähnten Verbesserungen die Performance und Benutzerfreundlichkeit signifikant steigern. Die Kombination einerseits aus einer einzigen Sitzung, rekursive Datenverarbeitung und GraphQL in backend und anderseits Caching in Frontend stellt eine leistungsfähigere Lösung als die vorhandene REST-API, für den Zugriff auf komplexe Datenmodelle dar und trägt zur Optimierung von Datenzugriffsarchitekturen in modernen Anwendungen bei.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

JSON	Java Script Object Notation
TCP	Transmission Control Protoco
XML	Extensible Markup Language
URI	Uniform Resource Identifier
SPA	Single Page Applications
TLS	Transport Layer Security
http	Hypertext Transfer Protocol
https	Hypertext Transfer Protocol secure
GUID	Globally Unique Identifier
CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
JS	JavaScript
TS	TypeScript
API	Application Programming Interfaces
URL	Uniform Resource Locator

Glossar

Boolean	Element, das die Zustände wahr oder falsch annehmen kann
Framework	Programmiergerüst für Entwickler zur Programmierung
String	Endliche Folge von Zeichen aus einem festgelegten Zeichensatz
Subscription	Im Kontext dieser Arbeit ein Abonnement von Daten

1 Einleitung

1.1 Motivation, Problemstellung und Zielstellung

In der heutigen Industrie nimmt die digitale Transformation eine zentrale Rolle ein. Automatisierung, intelligente Fertigung und vernetzte Maschinen sind essenzielle Bestandteile moderner Produktionsprozesse. Hierbei spielt der OPC UA (Open Platform Communications Unified Architecture) -Standard eine Schlüsselrolle [42]. Als universelle Schnittstelle, ermöglicht er den sicheren und standardisierten Austausch von Informationen zwischen unterschiedlichen Geräten und Systemen [40]. Da es komplexe Datenmodelle bereitstellt und verwaltet, ist OPC UA besonders in industriellen Anwendungen vom großen Nutzen.

Mit der zunehmenden Verbreitung von Web-Technologien und browserbasierten Anwendungen wächst jedoch die Herausforderung, OPC UA nahtlos in moderne, benutzerfreundliche Webanwendungen zu integrieren. Client-seitige Webkomponenten, die in Frameworks wie Angular oder React entwickelt werden, können nicht direkt mit einem OPC UA-Server kommunizieren. Daher ist eine zwischengeschaltete Web-API erforderlich. Diese dient als Vermittler und baut eine Brücke zwischen der Webanwendung und dem OPC UA-Server. Sie fungiert gegenüber dem OPC UA-Server als Client (OPC UA Client) und gegenüber der clientseitigen Web-Komponente als Web-Service (z.B. in Form einer REST-basierten Web API).

Abbildung 1: Zusammenspiel zwischen Clientseitige Web Komponente, Web API und OPC UA-Server (Quelle: eigene Darstellung)

Damit die Nutzer der Webanwendung ein zufriedenstellendes Nutzungserlebnis haben, soll die Webanwendung zügig auf Anfragen reagieren und die Ergebnisse entsprechend visualisieren.

Die vorliegende Bachelorarbeit untersucht die Optimierungsmöglichkeiten für den Zugriff auf ein OPC UA-Modell aus einer clientseitigen Webanwendung. In einer beispielhaften Implementierung, bei der eine Web-Komponente mit einer REST-API für OPC UA verbunden ist, zeigt sich, dass das System nur sehr träge auf Nutzeranfragen reagiert. Diese Verzögerungen führen zu einem unbefriedigenden Nutzungserlebnis, sodass die Anwendung in ihrer aktuellen Form nicht als nutzerfreundlich bezeichnet werden kann.
Ziel der Arbeit ist es daher, Ansätze zu identifizieren und zu bewerten, die die Leistung und Reaktionsfähigkeit der Webanwendung bei der Interaktion mit OPC UA-Modellen verbessern.
Folgende Relevante Use-Cases werden betrachtet:

- Darstellung des Adressraumes eines OPC UA-Servers (alle Knoten und Verbindungen)
- Aktualisierung der Darstellung bei Änderung des Adressraumes z.B Änderung von Knoten
- Option: Filterung des Adressraumes (Darstellung ausgewählter Knoten und deren Verbindungen)

1.2 Aufbau der Arbeit

Im ersten Teil der Arbeit werden theoretische Grundlagen der OPC UA, REST-API und GraphQL-API sowie Angular vorgestellt. Diese sind unterscheidend für das Verständnis der Arbeit.
Danach wird eine Ist- und Sollzustand Analyse durchgeführt um die Ziele der Arbeit durchzuleuchten und anschließend wird ein Optimierungskonzept entwickelt. In der Implementierung wird die Umsetzung der ausgewählten Lösung präsentiert. Der letzte Teil umfasst die Evaluation, das Fazit und ein Ausblick auf weitere mögliche Entwicklungen.

2 Theoretische Grundlagen

2.1 Open Platform Communications-Unified Architecture (OPC UA)

OPC steht für Open Platform Communications und ist eines der wichtigsten Kommunikationsstandards für die Industrie 4.0 und das IoT [42]. Mit OPC wird der Zugriff auf Maschinen, Geräte und andere Systeme im industriellen Umfeld standardisiert und ermöglicht den gleichartigen, herstellerunabhängigen und Plattformunabhängigen Datenaustausch in der Automatisierungs-technik [1]. Das UA in OPC UA steht dabei für "Unified Architecture" und bezeichnet die neuste Spezifikation des Standards [1]. Diese bewältigt die Herausforderungen moderner und vernetzter Systeme. Somit soll OPC UA die Interoperabilität zwischen verschiedenen Systemen ermöglichen und verbessern [2]. Die Interoperabilität wird durch standardisierte Transportprotokolle wie OPC TCP und HTTP(S) in Kombination mit Kodierungsmethoden wie OPC Binary, OPC XML und OPC JSON sichergestellt [3, S.3].
Die Kommunikation per OPC UA erfolgt nach dem verbindungsorientierten Client-Server-Modell [3, S.3].

Abbildung 2: Kommunikation zwischen OPC UA Client und OPC UA-Server (Quelle: eigene Darstellung)

OPC UA-Server stellt Daten und Dienste zur Verfügung, die von den Clients abgefragt oder genutzt werden können. Um Daten effizient an die Clients zu übertragen, verwaltet der Server das Informationsmodell, überwacht Zustandsänderungen und stellt Mechanismen zur Verfügung.
OPC UA Client fordert Daten oder Dienste an und interagiert mit dem Server. Der Client kann beispielsweise Werte lesen, schreiben oder Ereignisse abonnieren.

Die Kommunikation zwischen Client und Server im OPC UA-Protokoll kann sitzungsbasiert (Session) oder sitzungslos (Sessionless) erfolgen.
Beide Kommunikationsarten verwenden einen SecureChannel, um die Vertraulichkeit und Integrität der ausgetauschten Nachrichten zu gewährleisten [43] [9].
Die Dienste, die ein Client bei Bedarf vom Server anfordern kann, sind in Tabelle 1 aufgeführt.

Tabelle 1: OPC UA Dienste (Quelle: [44])

Service Set	Funktionsbeschreibung
SecureChannel Service Set	Abfrage des Endpunkts und der Sicherheitskonfiguration (security mode) um eine sichere Verbindung aufzubauen
Session Service Set	Dient zum Erstellen und Verwalten von anwenderspezifischen Verbindungen zwischen den Applikationen.
NodeManagement Service Set	Dient zum Modifizieren des Adressraums eines Servers, falls dieser das zulässt
View Service Set	Navigieren, folgen von (hierarchischen) Referenzen im Adressraum des Servers, Suchen und Filtern von Informationen
Attribute Service Set	Lesen und schreiben von Attributen einzelner oder mehrerer Knoten, vor allem das Value-Attribut, aber auch historische Daten oder Events.
Method Service Set	Aufrufen von Methoden (invoke), die ein Server an seinen Knoten im Adressraum anbietet
MonitoredItem Service Set	Dient zur Einstellung der Attibute von Knoten, die der Server überwachen und dessen Änderungen er melden soll
Subscription Service Set	Dient zum Erzeugen, Modifizieren oder Löschen von überwachten Items.
Query Service Set	Dient zur gefilterten Suche nach Informationen im Serveradressraum.

Ein OPC UA-Client nutzt Dienste wie Read, Write und Browse, um auf die Daten des Servers zuzugreifen. Alle diese Daten sind in einem hierarchisch aufgebauten Adressraum organisiert, der die Struktur und Zusammenhänge der Informationen abbildet [4].

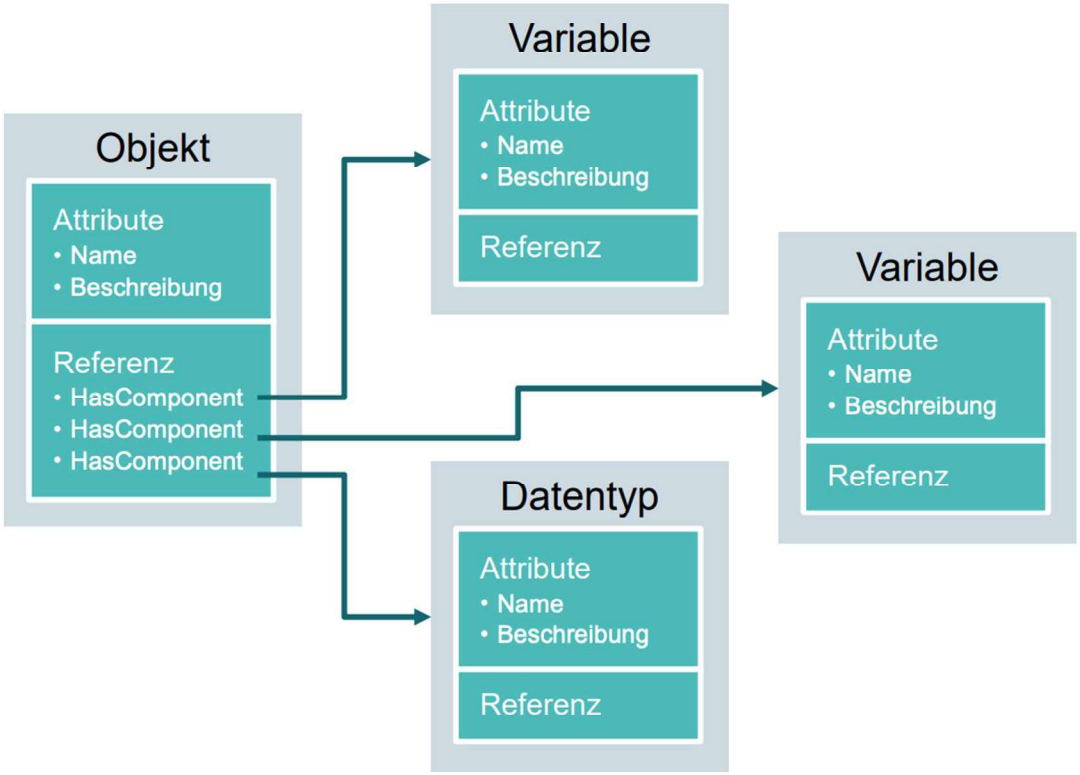


Abbildung 3: Adressraum eines OPC UA (Quelle: [4, S.9])

Die einzelnen Komponenten innerhalb dieser Struktur werden als Knoten (Nodes) bezeichnet und sind durch Referenzen miteinander verbunden. Jeder Knoten in einem Adressraum besitzt diverse Attribute, welche die Eigenschaften des Knotens beschreiben [4, S.9].

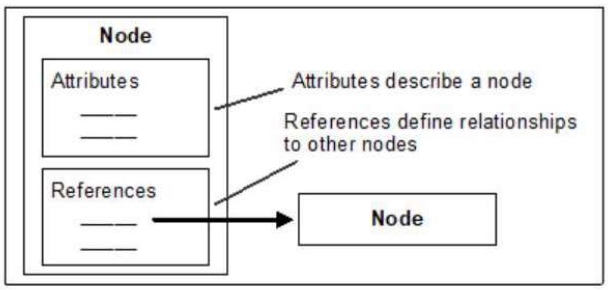


Abbildung 4: Schematische Darstellung des Node Model (Quelle: [8])

Attribute beschreiben die Eigenschaften eines Nodes. In Tabelle 2 sind die Attribute zusammengefasst, die allen Knotenklassen gemeinsam sind.

Tabelle 2: Zusammenstellung der Attribute (Quelle: in Anlehnung an [4, S.12])

Attribut	Beschreibung
NodeID	Die eindeutige Kennung der Node im Adressraum.
NodeClass	Gibt die Klasse der Node an (z. B. Objekt, Variable, Methode, Datentyp, Referenztyp, View usw.).
BrowseName	Ein lesbarer Name, der zum Browsen im Adressraum verwendet wird.
DisplayName	Ein benutzerfreundlicher Name
Description	Eine optionale Beschreibung der Node.
WriteMask	Gibt an,welche Attribute durch den Benutzer oder Client geändert werden können.
UserWriteMask	Ähnlich wie WriteMask, berücksichtigt aber Benutzerrechte und definiert, welche Attribute für einen bestimmten Benutzer schreibbar sind.

Das NodeId-Attribut wird für die eindeutige Identifizierung des Knotens innerhalb des OPC UA-Servers genutzt. Diese Node ID setzt sich aus einem Namespace zur Unterscheidung von Kennungen aus verschiedenen Subsystemen und einer Kennung, die entweder ein numerischer Wert, ein String oder eine GUID sein kann, zusammen [4 S.11].

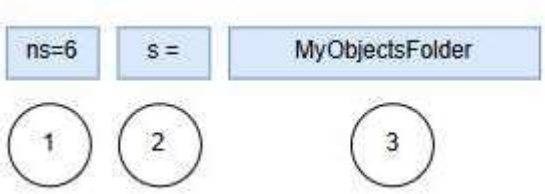


Abbildung 5: Bespiel einer NodeId in OPC UA (Quelle: Eigene Darstellung in Anlehnung an [4, S.11])

Tabelle 3: Aufbau einer NodeId (Quelle: in Anlehnung an [4, S.11])

1	Namespace Index
2	Node ID-Typ (s=String; i=Numerisch; g=GUID)
3	Identifier

OPC UA findet Anwendung in der Fertigungsindustrie, der Prozessautomatisierung, der Gebäudeautomatisierung und der Energieversorgung. Aufgrund seiner Fähigkeit zur nahtlosen Integration von Systemen und die Übertragung von Daten in Echtzeit, wird es zunehmend auch in IIoT-Umgebungen und Cloud-Anwendungen eingesetzt.

Subscription in OPC UA

Die Unterstützung von Subscription in OPC UA ist ein sehr wichtiges Konzept. Es ermöglicht Änderungen an bestimmten Datenpunkten in Echtzeit vom Client zu überwachen. [35]. In Anwendungen für Prozessüberwachung oder industrielle Fertigung wo Echtzeit Informationen entscheidend sind, ist dieser Konzept besonders wichtig.

Die Clients können Subscriptions in OPC UA erstellen, um Benachrichtigungen über Änderungen an Variablen oder Objekten in einem definierten Intervall zu erhalten und zu überwachen [45]. Der Vorteil in diesem Fall ist, dass der Client nicht diese Daten ständig abfragen muss. Dies reduziert die Netzwerklast und verbessert die Effizienz.

OPC UA-Subscriptions basieren auf einem Publisher-Subscriber-Modell. Ein Client (Subscriber) kann sich für bestimmte Datenpunkte (Variablen) anmelden. Wenn sich die Werte dieser Datenpunkte ändern, sendet der OPC UA-Server (Publisher) Benachrichtigungen an den Client. Diese Architektur ermöglicht eine flexible und skalierbare Kommunikation zwischen verschiedenen Systemen [45].

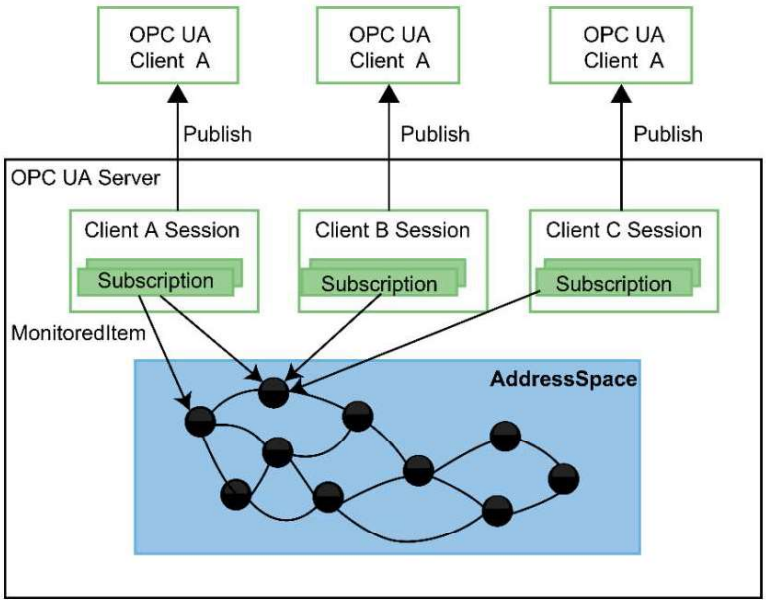


Abbildung 6: Subscription in OPC UA (Quelle: [46, S.3])

2.2 Representational State Transfer API (REST-API)

2.2.1 Definition

REST ist ein Architekturstil, der von Roy Fielding im Jahr 2000 in seiner Dissertation eingeführt wurde [7]. REST bietet einen flexiblen Ansatz für die Entwicklung von Webdiensten und ermöglicht dadurch unterschiedliche Interpretationen bei der Umsetzung [5]. Als Programmierparadigma hat sich REST aufgrund seiner Einfachheit und Skalierbarkeit zu einem der am weitesten verbreiteten Standards für APIs entwickelt.

2.2.2 Grundprinzipien der REST-API-Architektur

REST-APIs folgen sechs grundlegenden Prinzipien, die als RESTful-Prinzipien bekannt sind:

- Client Server Architektur [10, S.2]: In der REST-API-Architektur basiert die Interaktion zwischen Client und Server auf dem Prinzip der Client-Server-Trennung. Der Client ist für die Darstellung der Benutzeroberfläche und die Initiierung von Anfragen verantwortlich, während der Server die Anfragen verarbeitet und die erforderlichen Daten oder Dienste bereitstellt. Diese klare Trennung ermöglicht es, dass Client und Server unabhängig voneinander entwickelt, aktualisiert und skaliert werden können.
- Caching: REST-APIs unterstützen Caching. Antworten vom Server können zwischengespeichert (cached) werden. Dies verbessert die Effizienz und Leistung der Anwendung. Bei einer identischen Client Anfrage, kann der Cache die zuvor gespeicherte Antwort schnell zurückliefern d.h., die Anfrage wird nicht erneut an den Server gesendet. Dies reduziert die Latenz und die Serverlast, da weniger Anfragen verarbeitet werden müssen.
- Einheitliche Schnittstelle (Uniform Interface): die Vereinheitlichung der Schnittstellen ist eine der Haupteigenschaften von REST, weil sie die Architektur vereinfacht. Dies umfasst:
    - Ressourcen: Jede Ressource in einer REST-API hat eine eindeutige URI. Dies ermöglicht, die Ressource eindeutig zu identifizieren und darauf zuzugreifen.
    - HTTP-Methoden: Standardisierte HTTP-Methoden wie GET, POST, PUT und DELETE.
- Ressourcenorientierung: Ressourcen können in verschiedenen Formaten dargestellt werden, wie JSON oder XML. Dies ermöglicht Clients, die für sie am besten geeignete Repräsentation zu wählen.

- Schichten Architektur (Layered System): Eine REST-API kann aus mehreren Schichten bestehen, die jeweils unabhängig agieren. Diese Schichten können unterschiedliche Funktionen erfüllen, wie beispielsweise Lastverteilung, Authentifizierung oder Datenverarbeitung. Diese Modularität ermöglicht eine bessere Wartbarkeit und Skalierbarkeit der Anwendung.
- Zustandslosigkeit (Statelessness): Jeder Request, den der Client an den Server sendet, muss alle vom Server benötigten Informationen enthalten, um die Anfrage zu verarbeiten. Der Server speichert keinen Zustand zwischen den Anfragen. Dies fördert die Skalierbarkeit, da jeder Server unabhängig operieren kann, ohne Informationen über vorherige Anfragen zu benötigen.

2.2.3 Funktionsweise von REST-APIs

REST betrachtet Daten als Ressourcen, die über eindeutige URIs adressierbar sind. Die grundlegenden CRUD-Operationen (Create, Read, Update und Delete) werden mit HTTP-Metoden (z. B. GET, POST, PUT, DELETE) eingesetzt um die Interaktion mit diesen Ressourcen zu machen [10, S.2]. Die Kommunikation folgt dem Request-Response-Prinzip. Eine REST-API kann Ressourcen auf einem Server ansprechen und deren Zustände ändern. Ressourcen werden durch eindeutige URIs identifiziert und in standardisierten Formaten wie JSON oder XML übertragen [10, S.2].

Tabelle 4: Abbildung der HTTP-Verben auf die CRUD-Methoden (Quelle: eigene Darstellung)

Operation	**HTTP-Methode**	**Beschreibung**
Create (Erstellen)	POST	Anlage einer neuen Ressource
Read (Lesen)	GET	Abruf einer bestehenden Ressource
Update (Aktualisieren)	PUT	Modifikation einer Ressource oder Erstellung, wenn nicht vorhanden
Delete (Löschen)	DELETE	Entfernung einer bestehenden Ressource

Ein wichtiger Aspekt der Funktionsweise von REST-APIs sind die HTTP-Statuscodes, die die Antwort des Servers auf eine Anfrage kennzeichnen. Diese Codes informieren den Client über den Erfolg oder das Scheitern der Anfrage. Beispiele für häufig verwendete Statuscodes sind:

- 200 OK: Die Anfrage war erfolgreich, und die Antwort enthält die angeforderten Daten.
- 201 Created: Eine neue Ressource wurde erfolgreich erstellt.

- 204 No Content: Die Anfrage war erfolgreich, es gibt jedoch keine Daten zurück (z.B. nach einem DELETE).
- 400 Bad Request: Die Anfrage war fehlerhaft oder unvollständig.
- 404 Not Found: Die angeforderte Ressource konnte nicht gefunden werden.
- 500 Internal Server Error: Ein unerwarteter Fehler ist aufgetreten, der die Verarbeitung der Anfrage verhindert hat.

Die Verwendung dieser Statuscodes erleichtert die Kommunikation zwischen Client und Server, da sie eine schnelle und eindeutige Rückmeldung über den Status der Anfrage ermöglicht.

2.2.4 Vorteile von REST-APIs

- Einfachheit: REST-APIs sind leicht zu verstehen und zu implementieren, da sie auf bewährten Webstandards basieren.
- Skalierbarkeit: Die Zustandslosigkeit von REST ermöglicht die einfache Skalierung von Anwendungen.
- Flexibilität: REST-APIs unterstützen verschiedene Datenformate wie JSON, XML und HTML, was sie vielseitig einsetzbar macht.
- Plattformunabhängigkeit: REST-APIs können von jeder Plattform genutzt werden, die HTTP unterstützt.

2.2.5 Anwendungsgebiete

- Webanwendungen: Die Verbindung zwischen Frontend und Backend von Webanwendungen wird durch REST-APIs gestattet. Dies ermöglicht die Bereitstellung von dynamischen Inhalten und die Verarbeitung von Benutzereingaben.
- Mobile Anwendungen: REST-APIs werden von viele Mobile Apps verwendet, um Daten von Servern abzurufen oder zu aktualisieren.
- IoT (Internet of Things): Beim Datenaustausch zwischen IoT-Geräten und Cloud-Backends, spielen REST-APIs eine zentrale Rolle.
- Cloud-Dienste: Die Verwaltung und Interaktion mit Diensten von Plattformen wie Amazon Web Services (AWS) und Google Cloud wird mit REST-APIs angeboten.
- Integration von Drittanbietern: Integration von extern Dienste, z. B. Zahlungsanbieter oder Authentifizierungsdienste werden in bestehenden Anwendungen, durch REST-APIs, ermöglicht.

- Industrie 4.0: RESTful Kommunikation wird auch in industriellen Anwendungen eingesetzt. Sie gewährleistet die Interoperabilität zwischen Maschinen und IT-Systemen.

2.3 Graph Query Language (GraphQL) für OPC UA

2.3.1 Definition

GraphQL ist eine Abfragesprache [12] und gleichzeitig eine Ausführungs-Engine für APIs. Sie ermöglicht es Clients, genau die Daten zu spezifizieren, die sie benötigen [14]. Dies macht GraphQL besonders geeignet für komplexe Datenanforderungen. Die Sprache basiert auf einer hierarchischen Baumstruktur, die mit einem Wurzelknoten beginnt und sich in Kindknoten abspaltet [53]. GraphQL wurde 2012 von Facebook entwickelt und 2015 als Open-Source-Projekt veröffentlicht [13]. Aufgrund ihrer Flexibilität haben Unternehmen wie Airbnb, Booking und Netflix GraphQL in ihre Systeme integriert [15].

2.3.2 Schlüsselkomponenten von GraphQL

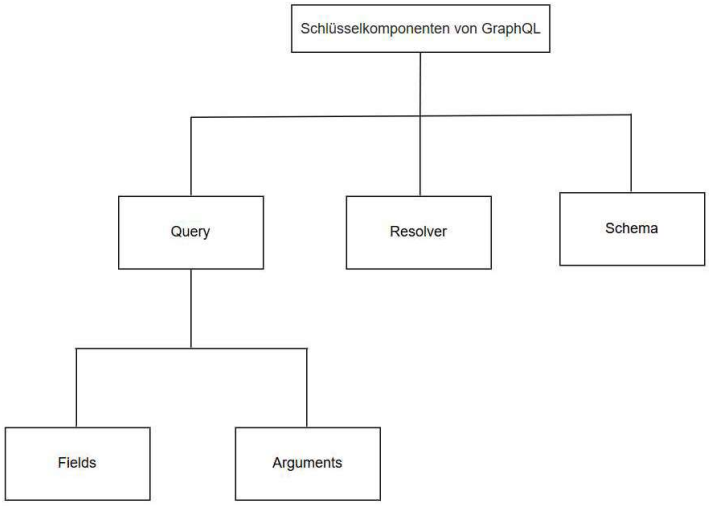


Abbildung 7: Schlüsselkomponenten von GraphQL

(Quelle: eigene Darstellung)

- **Schema**

Ein GraphQL-Schema ist das Herzstück einer GraphQL-API. Es definiert die Struktur der API. Es beschreibt welche Daten verfügbar sind, welche Typen und Beziehungen diese

Daten haben und welche Operationen (Abfrage, Mutation oder Subscriptions) unterstützt werden [16].

```
type Query {
  Sensors(id: ID!): Sensor
}

type Sensor {
  id: ID!
  type: String!
  status: String!
  location: Location!
}

type Location {
  roomName: String!
  floor: Int!
}
```

Listing 1: Beispiel eines GraphQL Schemas (Quelle: eigene Darstellung)

- **Query**

Eine GraphQL-Abfrage wird verwendet, um Daten von einem GraphQL-Server abzurufen. GraphQL erlaubt dem Client, genau die Daten zu definieren, die benötigt werden, und zwar in der gewünschten Struktur.

Tabelle 5: Elemente von GraphQL-Abfrage (Quelle: Eigene Darstellung in Anlehnung an [47])

Query Name	Gibt die Abfrageart an (z. B. Query, Mutation oder Subscription).
Felder	Gibt die spezifischen Felder an, die von einem Typ abgrufen werden sollen.
Argument	Filter oder Parameter, um die Abfrage zu spezifizieren

Merkmale einer GraphQL-Abfrage:

- Flexibilität: Der Client fordert nur die Daten an, die er benötigt [48]. Das vermeidet unnötige Informationen und spart Datenübertragungsvolumen.

```
query {
  Sensors(id: "1") {
    id
    type
    status
    location {
      roomName
      floor
    }
  }
}
```

Listing 2: Beispiel von GraphQL Abfrage (Quelle: eigene Darstellung)

Hier sind id, type, status und location die Felder des Typs Sensors. roomName und floor sind Unterfelder.

- **Antwort**

Eine GraphQL-Response ist die Antwort, die der Server nach einer erfolgreichen oder fehlerhaften GraphQL-Anfrage zurückgibt.

Eigenschaften der GraphQL-Antwort:

- Standardisiertes JSON-Format: Die Antwort ist leicht zu lesen und in Clients zu verarbeiten [18].
- Teilweise Ergebnisse: Selbst bei Fehlern in der Abfrage können verfügbare Daten zurückgegeben werden [18].

```
{
  "data": {
    "Sensors": {
      "id": "1",
      "type": "Temperature",
      "status": "Active",
      "location": {
        "roomName": "Server Room",
        "floor": 1
      }
    }
  }
}
```

Listing 3: Bespiel von GraphQL-Response (Quelle: eigene Darstellung)

- **Resolver**

Resolver in GraphQL sind Funktionen, die eine Antwort auf eine GraphQL-Abfrage generieren. Sie fungieren als Brücke zwischen der Anfrage des Clients und der Datenquelle (Datenbank oder API) [6]. Resolver müssen die Daten abrufen und in das erforderliche Format umwandeln, bevor sie an den Client gesendet werden.

Hauptmerkmale von GraphQL-Resolvern:

- Resolver werden für jeden benötigten Feldtyp im Schema definiert und liefern die entsprechenden Daten zurück.
- Argumente: Jeder Resolver akzeptiert vier Argumente [48]
  - parent: Enthält das Ergebnis des übergeordneten Resolvers.
  - args: Argumente, die in der Abfrage übergeben wurden.
  - context: Ein gemeinsam genutztes Objekt für alle Resolver.
  - info: Enthält Informationen über den Ausführungszustand der Abfrage.
- Asynchrone oder syncrone Ausführung: Resolver können asynchrone Operationen durchführen, z.B. Datenbankabfragen oder API-Aufrufe.
- Skalierbarkeit: Durch die Verwendung von Resolvern kann GraphQL effizient und flexibel Daten aus verschiedenen Quellen abrufen und manipulieren.

```
const resolvers = {
  Query: {
    Sensors: (_, { id }) => sensorsData.find(sensor => sensor.id === id),
  },
};
```

Listing 4: Beispiel von GraphQL-Resolver (Quelle: eigene Darstellung)

2.3.3 Die Hauptgestaltungsprinzipien von GraphQL [19]

- Hierarchische Struktur: Die Abfrage in GraphQL wird in einer Hierarchische Struktur definiert. Dies entspricht den Beziehungen und der Struktur der Daten in der API. Die Struktur der GraphQL-Antwort entspricht direkt der Struktur Ihrer Abfrage d.h. haben Abfrage und Antwort eine ähnliche Form.
- Produktzentriert: GraphQL ist aufgebaut auf die Anforderungen und Bedürfnisse von Front-End-Ingenieuren. Es orientiert sich an ihrer Denkweise und ihren Anforderungen und bietet eine Sprache sowie eine Engine, die genau dazu passt.
- Stark typisierte: GraphQL verwendet ein strenges Typsystem, um die Struktur der Daten klar zu definieren. Da der Typ jedes Felds im Schema festgelegt ist,

kann GraphQL sicherstellen, dass die Clients immer die richtige Art von Daten erhalten und die Server immer korrekte Daten verarbeiten.

- Client-spezifizierte Abfragen: Ein GraphQL-Server zeigt, welche Daten und Funktionen die Clients nutzen können. Ein Client definiert genau, welche Daten er benötigt, und gibt der Server im Gegensatz eine Antwort zurück, die der Struktur der Abfrage entspricht.
- Introspektiv: Man kann das Typsystem eines GraphQL-Servers direkt mit GraphQL selbst abfragen. Diese Möglichkeit macht es einfach, nützliche Tools und Bibliotheken für Clients zu erstellen [54].

2.3.4 Wie funktioniert GraphQL

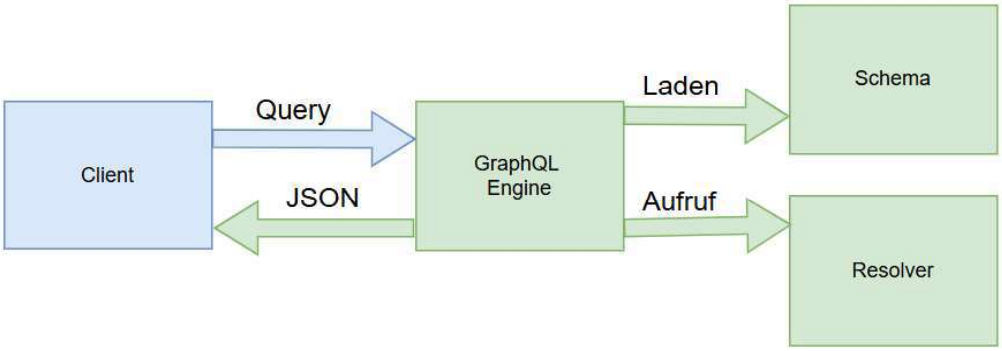


Abbildung 8: Die GraphQL-API Architektur

(Quelle: eigene Darstellung)

- Der Client sendet eine Anfrage an den Endpunkt des Servers und gibt genau an, welche Daten er benötigt. [20].
- Der GraphQL-Engine (Server) empfängt die Anfrage: das Schema wird geladen, um die Struktur der Daten zu verstehen und die Query wird an einen Resolver weitergeleitet. Der Resolver ist dafür zuständig, die Anfrage zu bearbeiten und die entsprechenden Daten abzurufen.
- Der Resolver holt die Daten vom Server: Der Resolver verarbeitet die Anfrage, ruft die nötigen Daten ab und bereitet die Antwort vor.
- Der GraphQL-Server sendet die formatierte Antwort in dem angeforderten Format (typischerweise als JSON) zurück an den Client.

2.3.5 Subscription in GraphQL

In GraphQL wird die subscription unterstützt, um Echtzeit-Updates von einem Server an Clients zu senden.

Mechanismus

In der GraphQL-Architektur wird eine Subscription in dem Schema durch die Definition eines speziellen Typs, eingerichtet. Clients können sich mit einer Subscription-Query an den Server anmelden damit sie auf bestimmte Ereignisse reagieren können. Der Server sendet automatisch eine Nachricht an alle angemeldeten Clients, sobald ein bestimmtes Ereignis eintritt (z.B. eine Änderung des Wertes eines überwachten Knotens) [50].

Implementierung

Die Implementierung von Subscriptions in einer API erfolgt häufig durch die Verwendung von WebSocket-Protokollen [50]. Mit WebSockets wird eine bidirektionale Kommunikation zwischen Client und Server ermöglicht. Der Server und auch der Client können jederzeit Daten senden und empfangen. Für Echtzeitanwendungen ist das sehr vorteilhaft, denn das verringert die Latenz und erhöht die Effizienz [49].
In GraphQL wird die Subscription mit WebSocket wie folgt realisiert [50]

- Schema-Definieren: Festlegen eines Subscription-Typs in GraphQL-Schema, um die zu abonnierende Ereignisse zu definieren.
- WebSocket-Server einrichten: Konfiguration einer WebSocket-Server für die Verarbeitung von Abonnements.
- Subscription-Resolver erstellen: Entwickelung von dem GraphQL Resolver, um das Abonnieren von Ereignissen zu ermöglichen.
- Veröffentlichen von Client-Events: Veröffentlichung der Ereignisse anhand des Pub/Sub-Mechanismus um die Clients über relevante Änderungen zu informieren. Wenn ein überwachendes Ereignis sich geändert hat, wird das Update durch pubsub.publish an alle aktiven Verbindungen gesendet.

2.4 Angular

Angular ist ein leistungsstarkes, open-source Webanwendungs-Framework, das von Google entwickelt wurde [21]. Entwicklern können mit Angular, dynamische, einseitige Webanwendungen (SPAs) erstellen, die eine gute Benutzererfahrung bieten [22]. Durch die Modulare Struktur von Angular, wird der Entwicklungsprozess vereinfacht, da spezifische Teile der Anwendung unabhängig voneinander entwickelt und getestet werden können und die Wartbarkeit der Anwendungen erhöht [23].
Angular basiert auf einer Komponenten-Architektur. Jede Anwendung besteht aus Modulen, die verschiedene Komponenten und Dienste gruppiert. Komponenten sind die Bausteine der Benutzeroberfläche. Sie enthalten sowohl die Logik als auch das Template (HTML und CSS) für einen bestimmten Teil der Anwendung. Daten können, Dank Dienste, von mehreren Komponenten genutzt werden. Diese Trennung fördert eine klare Struktur und Wiederverwendbarkeit [25].
Ein wichtiges Merkmal von Angular ist das Routing. In Angular können, verschiedene Ansichten innerhalb einer Anwendung navigieren, ohne die gesamte Seite neu zu laden. Die Nutzer können dann schnell zwischen unterschiedlichen Inhalten wechseln, ohne dass sie auf eine komplette Seitenaktualisierung warten müssen, und das verbessert die Benutzerfreundlichkeit. Entwicklern haben die Möglichkeit, dank des Routing-Moduls von Angular, Routen zu definieren und Parameter zu übergeben, was die Navigation intuitiv und flexibel macht [26].

```
const routes: Routes=[
  {path: 'Home', component : HomeComponent},
  {path: 'about', component: AboutComponent }
  // weitere Routen
];

@NgModule({
  declarations: [],
  imports: [
    RouterModule.forRoot(routes)
  ],
  exports:[RouterModule]
})
export class AppRoutingModule { }
```

Listing 5: Angular Routing-Konfiguration (Quelle eigene Darstellung)

Die zentrale Komponente im Angular Framework ist die „app.component.ts“. Diese fungiert als Root-Komponente und bildet die Grundlage für die gesamte Anwendung. Sie ermöglicht die Definition der Struktur der Benutzeroberfläche und integriert andere Komponenten innerhalb des Templates. Ihre zugehörigen Dateien, wie „app.component.html“ und „app.component.css“, ermöglichen die Gestaltung des Layouts und die Anwendung von Stilen. Die „app.module.ts“-Datei ist das Hauptmodule der Anwendung. Sie organisiert die verschiedenen Komponenten und deren Abhängigkeiten und stellt sicher, dass alle notwendigen Module importiert und die Komponenten korrekt deklariert sind [51]. Darüber hinaus enthält die Struktur nach Anwendung spezifische Unterverzeichnisse, die jeweils spezifische Komponenten oder Funktionalitäten repräsentieren.

Die Wahl des passenden Frameworks hängt von den spezifischen Anforderungen des Projekts und den Vorlieben der Entwickler ab.

Ein großer Vorteil von Angular ist die Typisierung durch TypeScript. Dies macht die Entwicklung sicherer [37].

3 Systematische Analyse

3.1 Istzustand

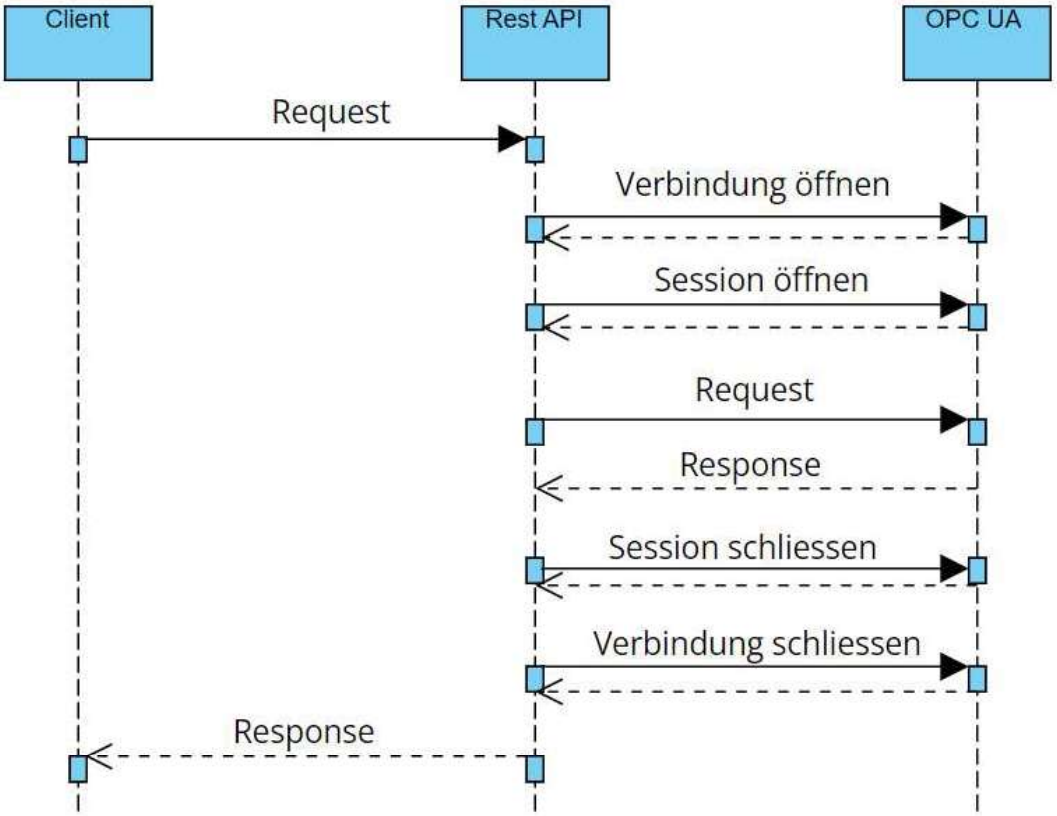


Abbildung 9: Istzustand Sequenzdiagramm (Quelle: eigene Darstellung)

Das Ziel der bestehenden Architektur, wie im Abbildung 1 dargestellt ist es, den Adressraum eines OPC UA-Servers zu visualisieren. Der Client sendet dazu eine HTTP (GET) Anfrage (Request) an die REST-API, um die Referenzen eines Knotens abzurufen. Die REST-API überprüft dann, ob bereits eine Verbindung zum OPC UA-Server besteht. Falls noch keine Verbindung existiert, wird eine neue Verbindung aufgebaut. Anschließend öffnet die REST-API eine Session mit dem OPC UA-Server, um eine sichere und strukturierte Kommunikation zu gewährleisten. Danach wird die Anfrage des Clients an den OPC UA-Server weitergeleitet. Der Server verarbeitet die Anfrage und generiert eine entsprechende Antwort in XML. Diese Antwort wird von der REST-API empfangen und umgewandelt in einem JSON-Format, um an den Client gesendet zu werden. Sobald die Anfrage vom REST-API empfangen wurde, wird die Session mit dem OPC UA-Server geschlossen, um Ressourcen freizugeben und dann

wird die Verbindung zum Server beendet. Schließlich wird die in JSON-Format vorbereitete Antwort an dem Client gesendet.

Der oben beschriebene Prozess wird im Frontend rekursiv wiederholt bis alle Knoten des Adressraumes dem Client (Frontend Webanwendung) zugestellt werden.

Performance-Analyse der bestehenden Lösung

Die vorhandene Implementierung des Systems ist stark optimierungsbedürftig, insbesondere in Bezug auf die Effizienz der Datenabfrage. Bei der aktuellen Nutzung der REST-API wird für jeden einzelnen Knoten eine separate Verbindung zum OPC UA-Server hergestellt und eine neue Sitzung geöffnet und nach Erhalt der Antwort geschlossen. Diese Vorgehensweise führt dazu, dass für n Knoten (wobei n die Gesamtanzahl der Knoten darstellt) n Verbindungen und n Sitzungen erstellt werden müssen. Dieser Prozess ist einerseits zeitaufwändig und anderseits belastet auch die Ressourcen des Servers erheblich.

Die folgende Abbildung zeigt die Request-Response Zeiten der bestehende REST-API für bestimmte Knoten.

```
node-opcua-api_rest: GET /api/0/nodes/ns%3D0%3Bi%3D32407/references 200 111.861 ms - 222 +163ms
node-opcua-api_rest: GET /api/0/nodes/ns%3D0%3Bi%3D32559/references 200 100.861 ms - 222 +963ms
node-opcua-api_rest: GET /api/0/nodes/ns%3D0%3Bi%3D32634/references 200 120.943 ms - 222 +189ms
node-opcua-api_rest: GET /api/0/nodes/ns%3D2%3Bi%3D4008/references 200 280.538 ms - 220 +408ms
node-opcua-api_rest: GET /api/0/nodes/ns%3D0%3Bi%3D23558/references 200 160.249 ms - 505 +482ms
node-opcua-api_rest: GET /api/0/nodes/ns%3D0%3Bi%3D17719/references 200 142.569 ms - 505 +489ms
]
```

Abbildung 10: Request Response Time für Vorliegende REST-API (Quelle: eigene Darstellung)

Die Performance-Messungen wurden auf einem Laptop durchgeführt, auf dem sowohl der OPC UA-Simulationsserver (Version 5.5.2) als auch die REST-API lokal liefen. Die REST-API wurde in Node.js (Version 18.12.1) mit dem Express-Framework (Version 4.18.2) implementiert. Da der OPC UA-Server und die REST-API auf demselben Gerät liefen, erfolgte die Kommunikation lokal ohne Netzwerklast. Die Tests wurden mit einer einzigen Client-Verbindung durchgeführt. Es wurden nur Leseoperationen (GET) auf dem OPC UA-Adressraum durchgeführt.

Erklärung der Log-Einträge im Abbildung 10

- API-Name „node-opcua-api_rest“: Dies ist der Dienst, der die Anfrage bearbeitet.
- HTTP-Methode „GET“: Diese Methode wird verwendet, um Daten abzurufen.
- URL-Pfad „/api/0/nodes/ns3D0%3Bi%3D32407/references“: Der spezifische Endpunkt für die Abfrage mit notwendigen Parametern.
- HTTP-Statuscode „200“: zeigt an, dass die Anfrage erfolgreich war.
- Antwortzeit „111,861 ms“: Die Zeit, die der Server benötigte, um die Anfrage zu verarbeiten.
- Weitere Zahlen:
  - „222“: Anzahl der Bytes in der Antwort.
  - „+163 ms“: Zusätzliche Zeit, die für die Verarbeitung oder den Netzwerkverkehr nach dem Senden der Antwort benötigt wurde (z.B. Latenz).

Die Analyse der Request-Response-Zeiten zeigt bei den vorliegenden REST-API, dass die Antwortzeiten für die Abfragen einzelner Beispielknoten zwischen etwa 270 ms (111,861+163) und 1060 ms (100,861+963) variieren. Diese Verzögerungen in den Antwortzeiten sind auf verschiedene Faktoren zurückzuführen, wie z.B. die Netzwerkbedingungen, die Serverlast, das Umwandeln der Antwort in JSON-Format oder auch die Komplexität der abgerufenen Knoten. Bei einer großen Anzahl von Knoten, wie sie in unserem Fall vorliegt, summieren sich diese Antwortzeiten erheblich, weil die Anfragen Sequenziell bearbeitet werden. Das System benötigt deswegen eine signifikante Zeitspanne für das vollständige Laden aller Knoten. Diese erheblichen Ladezeiten verdeutlichen die Ineffizienz der bestehenden Architektur und machen deutlich, dass eine Änderung in der Herangehensweise notwendig ist. Die aktuelle Situation führt einerseits zu längeren Wartezeiten für die Benutzer und andererseits auch zu einer erhöhten Serverlast, was letztendlich die gesamte Systemperformance negativ beeinflusst. Daher ist es entscheidend, alternative Ansätze zu untersuchen, um die Anzahl der Verbindungen und Sitzungen zu reduzieren und die Effizienz des Datenzugriffs zu verbessern.

3.2 Sollzustand

Der Soll-Zustand umfasst eine optimierte Architektur für die Datenabfrage und -visualisierung des OPC UA-Server-Adressraums. Ziel ist es, eine vollständige Darstellung aller Knoten und ihrer Verbindungen zu ermöglichen und dabei die Effizienz der Datenabfragen zu steigern. Bei Änderungen im Adressraum soll sich die Darstellung automatisch aktualisieren. Eine Filterfunktion wird zusätzlich bereitgestellt, um ausgewählte Knoten und Verbindungen anzuzeigen. Durch intuitive Funktionen wie Zoom und Suche wird die Benutzerfreundlichkeit verbessert. Nach Analyse und Optimierung der bestehenden Architektur wird die Umsetzung erfolgen, um eine leistungsfähige und benutzerfreundliche Lösung zu schaffen.

4 Optimierungskonzept

4.1 Untersuchung der SessionlessInvoke Methode für die REST-API

4.1.1 Definition der SessionlessInvoke Methode

SessionlessInvoke ist ein interessantes Konzept für REST-APIs, das in Version 1.04 der OPC UA-Spezifikation eingeführt wurde. Es ermöglicht zustandslose Anfragen an einen OPC UA-Server. Bei Sessionlessinvoke kann der Client mit dem Server kommunizieren, um bestimmte Dienste aufzurufen, ohne eine Session einzurichten. Dies umfasst Dienste wie Read, Write oder Call, die keine benutzerspezifischen Zustandsinformationen benötigen [9]. Traditionell erfordert OPC UA die Einrichtung eines sicheren Kanals und einer Sitzung. In die Sitzung werden statusrelevante Informationen gespeichert und das führt zu erhöhtem Overhead und Komplexität [10, S.4]. In Umgebungen mit vielen gleichzeitigen Clients, wie im Kontext des Industrial Internet of Things (IIoT), können diese Anforderungen problematisch sein, weil sie die Skalierbarkeit und Effizienz der Systeme beeinträchtigen.

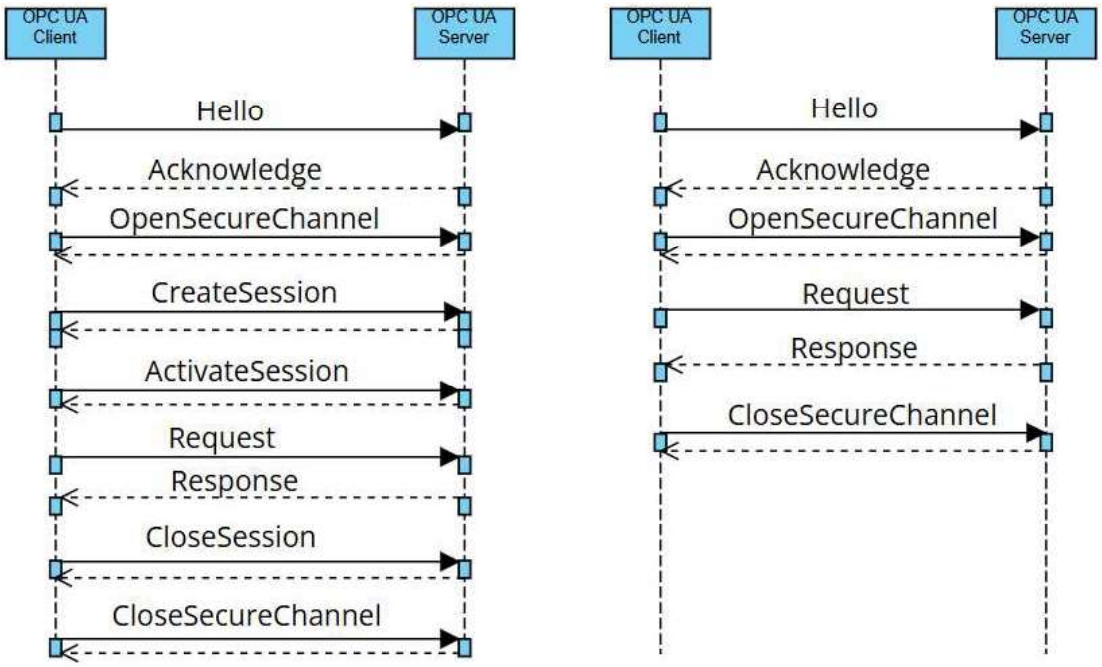


Abbildung 11: Vergleich von Sequenzdiagrammen zwischen sessionbasiertem und sessionless in einer REST-API für OPC UA (Quelle: eigene Darstellung in Anlehnung an [28])

Dieses vereinfachte Diagramm stellt ein Vergleich zwischen den Session-basierte und den Sessionlessinvoke Methoden dar.

Session-basierte Methode besteht aus mehreren Schritten. Zuerst wird eine CreateSessionRequest gesendet gefolgt von einer entschprechenden CreateSessionResponse. Nach der Erstellung der Session erfolgt die ActivateSessionRequest, worauf eine ActivateSessionResponse folgt. Der Client kann dann seine spezifischen Service-Requests, wie etwa ReadRequest, senden. Eine Session wird aufrechterhalten, bis sie geschlossen wird.
Bei der sessionless Kommunikation, im Gegensatz, entfällt die Sessionerstellung komplett. Der Client sendet direkt eine Anfrage, und erhält umgehend eine Antwort. Eine Session-Aktivierung oder einen Session-Abbau ist hier nicht notwendig [41, S.1].

4.1.2 Vorteile der SessionlessInvoke Methode

Der Sessionlessinvoke reduziert der Kommunikationsaufwand zwischen Client und Server [10, S.9] und bietet somit einige Vorteile. Die Notwendigkeit eines ständigen Datenaustauschs im Zusammenhang mit der Einrichtung und Aufrechterhaltung von Sitzungen entfällt aufgrund des Fehlens einer Sitzungsverwaltung. Clients können, mit der Einführung von "SessionlessInvoke", direkt auf OPC UA-Dienste zugreifen und in ihren Anfragen alle notwendigen Informationen mitliefern. Der Server muss in diesem Fall keine spezifischen Informationen für jeden Client speichern. Dies reduziert die Anzahl der zwischen den beiden Entitäten ausgetauschten Pakete, vereinfacht den Kommunikationsprozess und optimiert die Bandbreitennutzung. Der Ressourcenverbrauch der Server wird dadurch reduziert und die Verarbeitungsgeschwindigkeit erhöht [10, S.4].
Diese Eigenschaft ist ein Vorteil, insbesondere in Umgebungen, in denen Verarbeitungsgeschwindigkeit und Ressourceneffizienz Priorität haben.
Außerdem ermöglicht dieser Ansatz bei einfachen Anfragen eine hervorragende Skalierbarkeit, insbesondere für Server, die eine große Anzahl angeschlossener Geräte verarbeiten müssen. Der Server lässt sich problemlos auf eine steigende Anzahl von Clients oder Knoten skalieren, ohne durch die Sitzungsverwaltungskapazität eingeschränkt zu werden, da zwischen den Anforderungen keine Sitzungsinformationen gespeichert werden [10, S.5]. Besonders in komplexer Industrieumgebungen, in denen die Anzahl der miteinander verbundenen Geräte die Leistung des Servers nicht beeinträchtigen soll, ermöglicht diese Flexibilität eine Bessere Unterstützung.
Ein weiterer Vorteil des "SessionlessInvoke"-Mechanismus ist die Möglichkeit, Caching-Strategien zu implementieren. Dies kann die Effizienz der Datenübertragung weiter steigern [10, S. 6].

Diese Eigenschaften machen "SessionlessInvoke" zu einer potenziell interessanten Lösung für die Optimierung der Kommunikation in industriellen Anwendungen.

4.1.3 Diskussion und Analyse der Effizienz

Für komplexe Umgebungen bietet SessionlessInvoke mehrere Vorteile, jedoch einer der größten Nachteile dieser Methode in unseren bestehenden Rest-API ist, dass für jede Anforderung für jeden Knoten im System eine neue sichere Verbindung (OpenSecureChannel) hergestellt werden muss.
Dies bedeutet, dass für jede Anfrage eine separate Verbindung hergestellt werden muss und das kann schnell zu einer extrem hohen Anzahl gleichzeitiger Verbindungen führen. Wenn beispielsweise 1000 Knoten abgefragt werden müssen, werden 1000 separate Verbindungen generiert, eine pro Knoten. Diese große Anzahl an Verbindungen führt zu einer komplexen Verwaltung und kann zu einer Netzüberlastung führen.
Für jede neue Verbindung müssen mehrerer Initialisierungsnachrichten wie „Hallo“ (Connect) und „Bestätigung“-Nachrichten ausgetauscht werden, die zum Herstellen und Validieren der Verbindung erforderlich sind. Zusätzlich werden TLS-Handshake Nachrichten ausgetauscht um eine Sichere Kanal (SecureChannel) herzustellen, bevor Daten übertragen werden können [52]. Diese Schritte sind notwendig, weil sie die Sicherheit und Integrität des Austauschs gewährleisten. Gleichzeitig erhöhen sie aber den Bandbreitenverbrauch und die Verarbeitungszeit, weil erst jeder Austausch gesendet und empfangen werden muss und dann beginnt die eigentliche Transaktion. In Systemen mit einer großen Anzahl von Knoten oder mit hohem Datenverkehr kann dieser Overhead erhebliche Auswirkungen auf die Gesamteffizienz des Systems haben. Der Betrieb wird verlangsamt und die Gesamtleistung verringert durch das Verwalten und Übertragen dieser ersten Nachrichten, weil so entsteht eine zusätzliche Latenz. Wenn eine große Anzahl gleichzeitiger komplexe Anforderungen schnell verarbeitet werden muss, dann wird dies besonders problematisch.
Als Zusammenfassung kann man sagen, dass das Sitzungslose Modell hat als Vorteil die Vereinfachung des Sitzungsverwaltungs, hat aber den gegenteiligen Effekt, das Netzwerk bei komplexen Abfragen zu überlasten und die Interaktionen mit dem Server zu verlangsamen.
Außerdem wird diese Methode von der Prosys OPC UA Simulation Server nicht unterstützt, deswegen stellt sie, in unserem Anwendungsfall keine sinnvolle Lösung dar, um die Systemleistung zu verbessern. Stattdessen sollten sitzungsbasierte Ansätze mit weiteren Optimierungen in Betracht gezogen werden, um die Effizienz in komplexen Umgebungen zu steigern.

4.2 Untersuchung einer Methode mit einer einzigen Session

In der Vorliegenden API stellt die Zeitverzögerung, die durch die Erstellung einer Sitzung für jeden einzelnen Knotenaufruf entsteht, um alle Knoten im Frontend zu visualisieren, das Hauptproblem dar. Daher wird hier eine neue API untersucht, mit dem Ziel die Effizienz beim Zugriff auf das OPC UA-Datenmodell deutlich zu verbessern. Im Gegensatz zur vorherigen Implementierung der REST-API, soll diese neue API alle Knoten und deren Referenzen in einer einzigen Sitzung abrufen.

Wie in Abbildung 12 dargestellt, sendet der Client eine Anfrage an die API, um alle Knoten abzurufen. Daraufhin öffnet die API eine Verbindung zum OPC UA-Server und erstellt eine Sitzung. In dieser Sitzung wird ein Request an den Server gesendet, um die benötigten Informationen über alle Knoten anzufordern. Die API ruft alle relevanten Knotenaten und Referenzen in einem einzigen Durchlauf ab, wodurch die Anzahl der Anfragen und die damit verbundenen Latenzzeiten reduziert werden. Erst wenn alle Knoten erfolgreich abgerufen wurden, wird die Sitzung geschlossen und die Verbindung zum OPC UA-Server vom API getrennt. Diese Methode soll die Performance verbessern, weil die Zeit für die Eröffnung und Schließung mehrerer Sitzungen entfällt.

Diese neue API kann entweder mit REST oder mit GraphQL entwickelt werden.

Im nächsten Abschnitt wird ein detaillierter Vergleich zwischen GraphQL und REST durchgeführt, um eine fundierte Entscheidungsgrundlage zu schaffen, welche API-Architektur in unserem spezifischen Anwendungsfall besser geeignet ist.

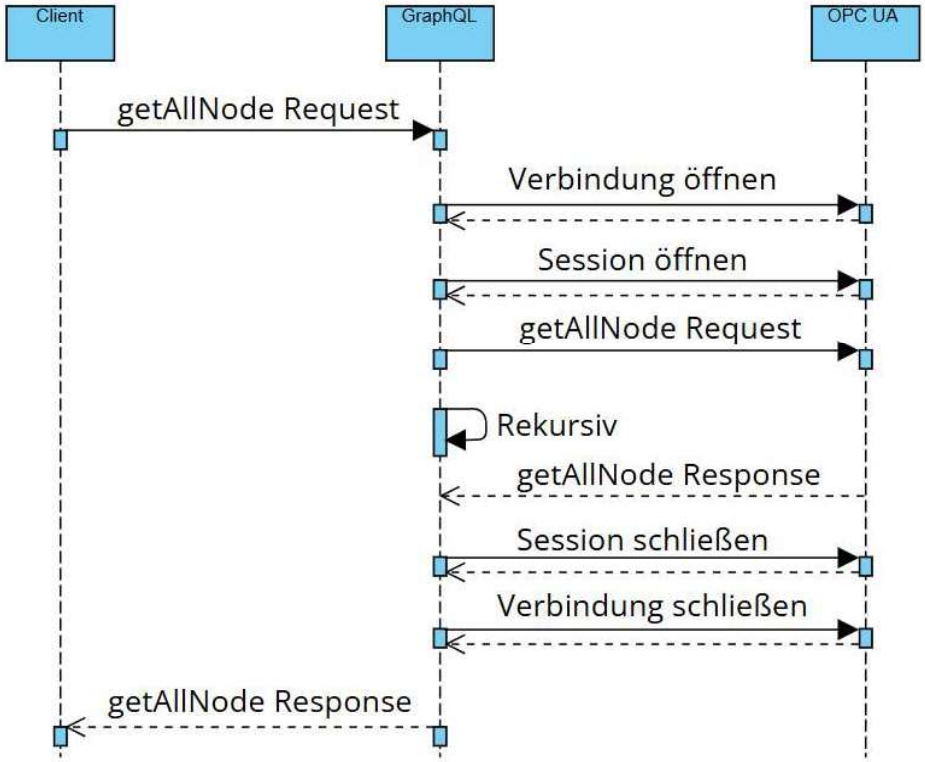


Abbildung 12: Sequenzdiagramm bei einer einzigen (Quelle: eigene Darstellung)

4.3 GraphQL als Alternative für REST

In diesem Abschnitt wird ein Vergleich zwischen GraphQL und Rest untersucht:

Tabelle 6: Unterschied zwischen GraphQL und REST-API (Quelle: in Anlehnung an [36])

	Rest	**GraphQL**
Communication Model	Client- Server	Client-Server und Subscription
Protocol	HTTP	HTTP
Cache	An verschiedenen Punkten	Anwendungsspezifisch
Scalability	Gut	Mittel
Interface	Uniform	Anwendungsspezifisch
Ease of Use/Development	Mittel/Gut	Gut/Gut
Performance	Niedrig	Mittel
Overfectching/ underfetching	Häufig	Selten

4.3.1 Kommunikationsmodell

- Rest folgt dem traditionellen Client-Server-Modell. Der Client sendet eine Anfrage an den Server, der daraufhin eine Antwort zurückgibt. Jede Interaktion erfolgt durch vollständige HTTP-Requests.
- GraphQL unterstützt zusätzlich zur klassischen Client-Server-Kommunikation auch das Modell der Subscriptions [29]. Mit dieser Funktion kann der Server, sobald sich Daten ändern, an den Client Echtzeit-Updates senden. Das ist besonders nützlich in Systemen, die mit ständig wechselnden Daten arbeiten müssen.

4.3.2 Protokoll

Beide Technologien verwenden HTTP als Protokoll, was sie mit modernen Web-Technologien kompatibel macht.

- REST: HTTP-Methoden (GET, POST, PUT, DELETE) werden verwendet, um auf die Ressourcen des Servers zuzugreifen.
- GraphQL: Hier wird ausschließlich HTTP verwendet, aber anstelle von festen Endpunkten für jede Ressource gibt es einen zentralen Endpunkt für alle Datenanfragen [30].

4.3.3 Cache

Ein großer Vorteil von REST gegenüber GraphQL ist das einfache Caching.

- Rest: Bei REST kann das Caching an verschiedenen Punkten stattfinden. Jede REST-Anfrage enthält eine eindeutige URL und kann der Cache mit diesem eindeutigen Identifikator verknüpft werden [36, S.7].
- GraphQL: GraphQL bietet keine integrierten Caching-Mechanismen. In diesem Fall Caching ist aber auch möglich mit Verwendung eine GraphQL Client wie Apollo. GraphQL-Clients bieten integrierte Caching-Mechanismen die Abfragen auf Clientseite zwischenspeichern können [32].

4.3.4 Skalierbarkeit

- Rest: Aufgrund seiner Architektur lässt sich REST leicht skalieren und Lasten zwischen mehreren Servern verteilen. Außerdem kann es von verschiedenen Caching-Strategien profitieren, um die Leistung zu optimieren [33, S.7].

- GraphQL: GraphQL bietet eine mittlere Skalierbarkeit da Caching schwieriger umzusetzen ist. Zudem sind Abonnements nicht zustandslos und erfordern eine offene Verbindung zum Client, was Ressourcen bindet.
  GraphQL kann aber bei mehreren Clients besser performen als REST auch ohne Caching [33, S.7]. Seine Fähigkeit, mehrere Ressourcen in einer einzigen Anfrage abzufragen, reduziert die Anzahl der Anfragen, die an den Server gesendet werden müssen, und verbessert damit die Effizienz.

4.3.5 Interface

- Rest: Ein wesentlicher Aspekt von REST ist das Uniform Interface. Jede Ressource ist über definierten Endpunkt zugänglich. Durch dieses Interface sollte diese Ressource die Möglichkeit bieten, verwandte oder zusätzliche Daten mit HTTP-Methoden (GET, POST, PUT oder DELETE) abzurufen [7].
- GraphQL: GraphQL ist Anwendungsspezifisch und lässt sich flexibel an die Anforderungen der Anwendung anpassen. Der Client kann in einer einzigen Anfrage angeben, welche Daten er benötigt. Es stehen drei Arten von Operationen zur Verfügung: Query, Mutation und Subscriptions.

4.3.6 Ease of use/Development

- REST: REST ist mittelmäßig einfach zu nutzen und zu entwickeln. Die API-Endpunkte müssen manuell definiert und dokumentiert werden, und der Entwickler muss sich auf die Struktur der Daten fokussieren. Es erfordert auch mehr Planung und Aufwand beim Entwickeln und Pflegen der API.
- GraphQL: GraphQL gilt als entwicklerfreundlich und bietet durch Tools wie GraphiQL[34] eine hohe Benutzerfreundlichkeit. Entwickler können mithilfe der Introspektion des GraphQL-Schemas sofort sehen, welche Daten verfügbar sind und wie sie abgefragt werden können. Außerdem erleichtert es das Schreiben und Testen von Abfragen.
  In GraphQL wird außerdem mit einem einzigen Endpunkt gearbeitet und dies vereinfacht die Client-Integration. Entwickler müssen sich nicht um die Vielzahl der REST-Endpunkte kümmern, sondern können alle Anfragen an eine einzige URL senden.

4.3.7 Performance

- REST: REST bietet niedrigere Performance bei mehreren Anfragen, da für jede Ressource eine eigene Anfrage gestellt werden muss. Auch bei sehr detaillierten Abfragen, bei denen unnötige Daten geladen werden, kann dies ineffizient sein.
- GraphQL: GraphQL bietet mittlere Performance, da nur die benötigten Daten abgerufen werden können und daher Bandbreite eingespart wird. Bei komplexeren Abfragen oder bei gleichzeitigen Anfragen an mehrere Datenquellen kann die Performance jedoch beeinträchtigt werden, wenn die Server-Infrastruktur nicht optimal konzipiert ist.

4.3.8 Overfectching/ underfetching

- REST: In REST gibt es häufig Overfetching oder Underfetching, was bedeutet, dass entweder zu viele oder zu wenige Daten abgerufen werden. Beispielsweise könnte ein Client eine große Menge an Daten abfragen, obwohl er nur einen Teil davon benötigt.
  In REST können mehrere Anfragen erforderlich sein, um verwandte Daten zu fetchen.
- GraphQL: In GraphQL ist Overfetching oder Underfetching selten, da der Client genau festlegt, welche Daten er abruft. Dies führt zu einer effizienteren Nutzung der Bandbreite und einer geringeren Belastung des Servers.

4.3.9 Verbindung mit OPC UA

Im Artikel „Comparison of REST and GraphQL Interfaces for OPC UA“ [36] wurde eine Experimentanalyse zu dem Vergleich der Ausführungszeiten von REST-API und GraphQL in der Verbindung mit OPC UA durchgeführt. Die Antwortzeiten sind in folgender Abbildung dargestellt.

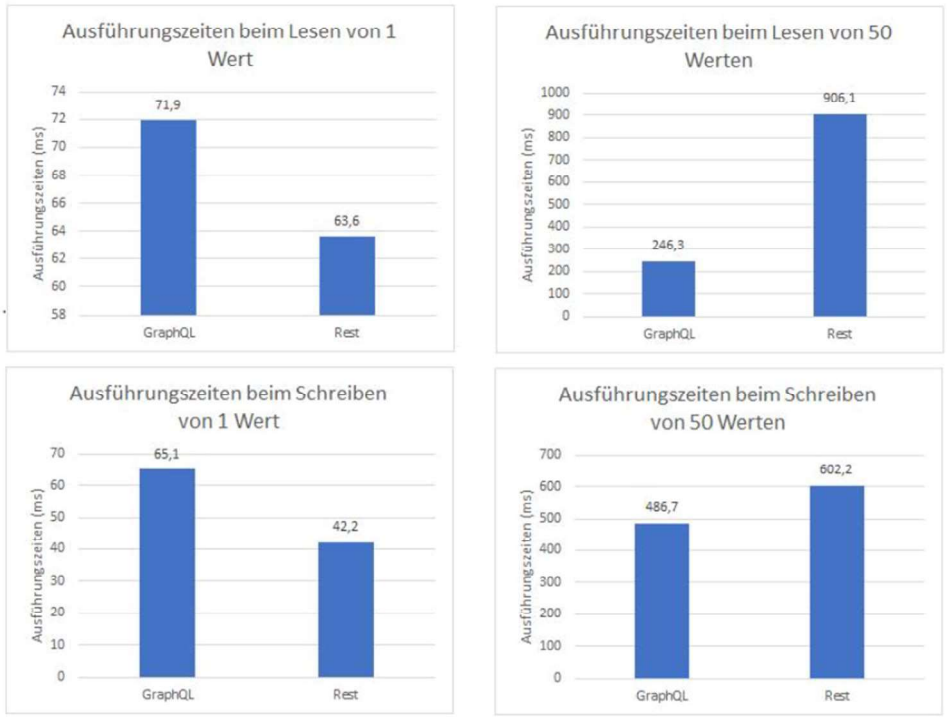


Abbildung 13: Vergleich der Ausführungszeiten im Falle von Lesen und Schreiben auf einem OPC UA-Server (Quelle: eigene Darstellung in Anlehnung an [36])

Die Bilder zeigen die Ausführungszeiten für das Lesen und Schreiben von Werten auf einem OPC UA-Server über verschiedene Schnittstellen (REST und GraphQL)
der Ausführungszeiten von REST und GraphQL zeigt Unterschiede in der Leistung, die auf die spezifischen Eigenschaften der jeweiligen Schnittstellen und die Art der Datenoperationen zurückzuführen sind. REST bietet eine bessere Leistung bei einfachen Lese- und Schreiboperationen, insbesondere bei einzelnen Werten. GraphQL hingegen bringt Vorteile beim Umgang mit komplexen Abfragen und größeren Datenmengen.

Diese Analyse zeigt, dass die Wahl der Schnittstelle auf die spezifischen Anforderungen der Anwendung abgestimmt werden soll. REST ist optimal für schnelle und einfache Abfragen, während GraphQL für komplexere Abfragen vorteilhafter sein kann. Bei Anforderungen an die Effizienz von Lese- und Schreiboperationen und einfache Abfragen sollte REST priorisiert werden. Für Anwendungen, die komplexe Datenstrukturen erfordern, kann GraphQL eine geeignete Option darstellen.

4.4 Caching

Ein zentraler Aspekt bei der Optimierung der Antwortzeit ist die Nutzung von Caching. In der Informationstechnologie ist ein Cache eine Hochgeschwindigkeitsspeicherebene, die eine Teilmenge von Daten temporär speichert, um zukünftige Anfragen schneller bedienen zu können. Durch Caching können bereits abgerufene oder berechnete Daten effizient wiederverwendet werden, was die Leistung von Anwendungen erheblich steigert [55].

GraphQL stellt eine Herausforderung für traditionelle Caching-Mechanismen dar. Es verwendet nur einen einzigen Endpunkt, und die Anfragen sind oft komplex und individuell [57, S.19]. Im Gegensatz zu REST, bei dem jede Ressource über einen eindeutigen URI identifiziert wird, sendet GraphQL alle Anfragen an denselben Endpunkt, was das Caching erschwert [57, S.20]. Standard-HTTP-Caching ist daher nicht direkt anwendbar, da jede Anfrage unterschiedliche Daten zurückgeben kann [57, S.20].

Um Caching in GraphQL zu ermöglichen, wird häufig das Apollo Client-Framework eingesetzt, das eine integrierte Caching-Lösung namens InMemoryCache bietet. Dieser Cache speichert die Ergebnisse von GraphQL-Anfragen in einer normalisierten Lookup-Tabelle, wodurch zukünftige Anfragen für dieselben Daten ohne erneute Netzwerkanfrage bedient werden können [59].

Zum Beispiel, wenn die Anwendung zum ersten Mal eine GetSensor-Abfrage für ein Sensor-Objekt mit der ID 123 ausführt, sieht der Ablauf folgendermaßen aus:

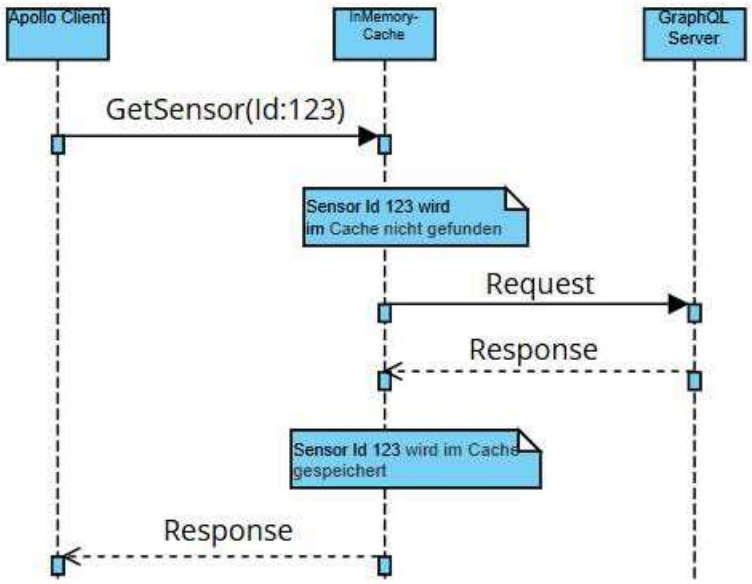


Abbildung 14: Erster Aufruf: Sensor-Daten werden vom Server abgerufen

(Quelle:eigene Darstellung in Anlehnung an [32])

Und jedes Mal, wenn die Anwendung später GetSensor für dasselbe Objekt ausführt, sieht der Ablauf stattdessen so aus:

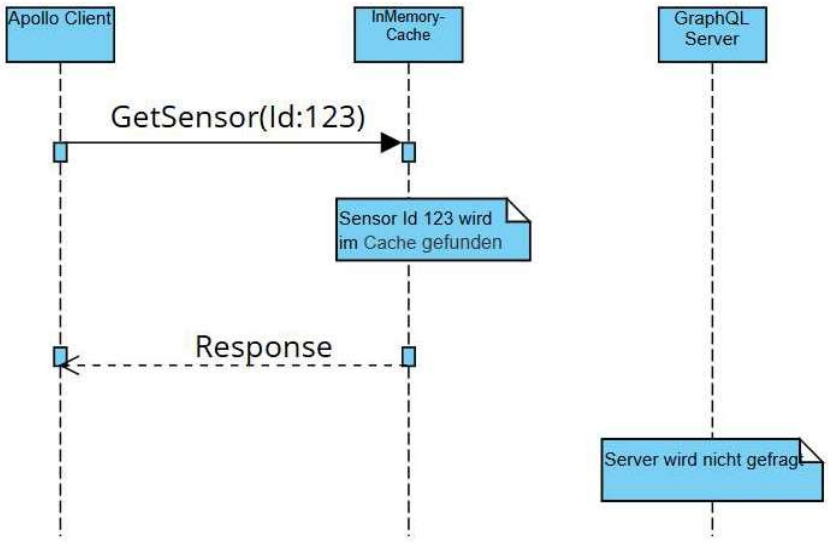


Abbildung 15: Folgende Aufrufe: Sensor-Daten werden aus dem Cache geladen (Quelle: eigene Darstellung in Anlehnung an [32])

Neben clientseitige Lösungen wie Apollo Client, können auch Serverseitige Lösungen, wie Apollo Server, Caching-Mechanismen (z.b. Redis oder Memcached) nutzen [62]. Trotz dieser Verbesserungen bleibt das Caching in GraphQL komplexer als in REST, da es keine nativen HTTP-Caching-Mechanismen nutzt. Entwickler müssen spezielle Bibliotheken und Techniken verwenden, um Caching effektiv zu implementieren, was zu einem höheren Implementierungsaufwand führen kann, insbesondere in Anwendungen, die stark auf Caching angewiesen sind [57, S.19].

4.5 Entscheidung für die Implementierung

Zu Beginn meiner Untersuchung zur Optimierung der Visualisierung des Adressraumes von OPC UA habe ich die Möglichkeit des Einsatzes von SessionlessInvoke in Betracht gezogen. Allerdings stellte sich heraus, dass diese Methode von Prosys OPC UA Simulation Server nicht unterstützt wurde. Dies führte mich zu der Überlegung, dass eine Implementierung mit einzelner Sitzung, wo alle erforderlichen Referenzen abgerufen werden, möglicherweise eine bessere Lösung darstellt.
Im Zuge meiner Recherchen habe ich einen Vergleich zwischen GraphQL und REST durchgeführt. Dabei konnte ich feststellen, dass GraphQL bessere Vorteile beim Abrufen mehrerer Datensätze bietet. Während REST mehrere Anfragen an unterschiedliche Endpunkte erfordert, ermöglicht GraphQL die Aggregation aller benötigten Daten in einer einzigen Anfrage. Diese Flexibilität ist besonders vorteilhaft, um komplexe

Datenstrukturen effizient zu bearbeiten, weshalb es mittlerweile eine beliebte Alternative zu REST [61].

Außerdem ist es zu bemerken, dass viele große Unternehmen auf GraphQL umgestiegen sind. Dieser Trend verdeutlicht die wachsende Anerkennung und den Nutzen von GraphQL in der Softwareentwicklung. Die Fähigkeit, geziehlte Daten nach Bedarf abzurufen, ohne überflüssige Informationen zu laden, ist für moderne Anwendungen ein entscheidendes Kriterium. Zudem, aufgrund der nativen Unterstützung von Subscriptions eignet sich GraphQL hervorragend, um Daten in Echtzeit zu überwachen.

Aufgrund dieser Erkenntnisse habe ich mich entschieden, die API unter Verwendung von GraphQL mit einer einzigen Sitzung zu implementieren. Zusätzlich plane ich, einen Caching-Mechanismus im Frontend zu integrieren. Diese Entscheidung wird meiner Einschätzung nach einer leistungsfähigen und flexiblen Schnittstelle für die Visualisierung des OPC UA-Adressraums ermöglichen. Durch die Kombination von GraphQL und Caching wird nicht nur die Effizienz der Datenabfragen verbessert, sondern auch die Benutzerfreundlichkeit der Anwendung deutlich gesteigert.

5 Implementierung

In diesem Abschnitt werden die verschiedenen Schritte der Implementierung vorgestellt und erläutert, beginnend mit dem Backend, also der API, gefolgt vom Frontend, sprich die Webanwendung.

5.1 Backend (API)

5.1.1 Anforderungsanalyse

Funktionale Anforderungen (FA) (Anhang 14)

- FA-01: Zugriff auf alle Knoten und Hierarchien in einer Abfrage (*Hoch*).
- FA-02: Verbindung zum OPC UA-Server verwalten (*Hoch*).
- FA-03: Unterstützung verschiedener Datentypen in GraphQL (*Mittel*).
- FA-04: Knotenstruktur bis zu einer definierten Tiefe filtern (*Mittel*).
- FA-05: Knotenstruktur ab beliebiger „nodeId“ abrufen (*Mittel*).
- FA-06: Echtzeit-Benachrichtigung bei Wertänderungen (*Hoch*).
- FA-07: Hohe Performance, auch bei großen Strukturen (*Hoch*).

Nicht-funktionale Anforderungen (NFA) (Anhang 15)

- NFA-01: Code-Qualität (Hoch).
- NFA-02: Standards & Frameworks (Hoch).
- NFA-03: Dokumentation (Mittel).
- NFA-04: Lizenzfreie Software (Hoch).

5.1.2 Implementierung der GraphQL-API

Die detaillierte Darstellung der Funktionen in Sequenzdiagramm verdeutlicht die Funktionsweise der API und zeigt, wie die verschiedenen Komponenten zusammenarbeiten, um die Daten effizient zu verwalten.

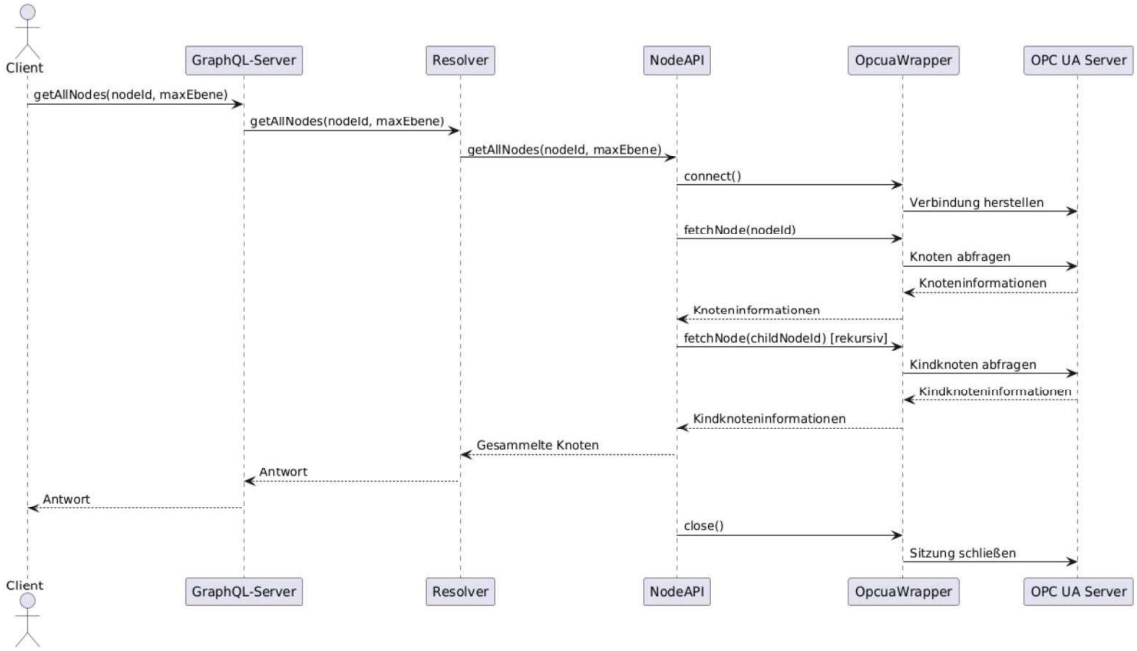


Abbildung 16: Sequenzdiagramm: Abfrage aller Knoten (getAllNodes) (Quelle: eigene Darstellung)

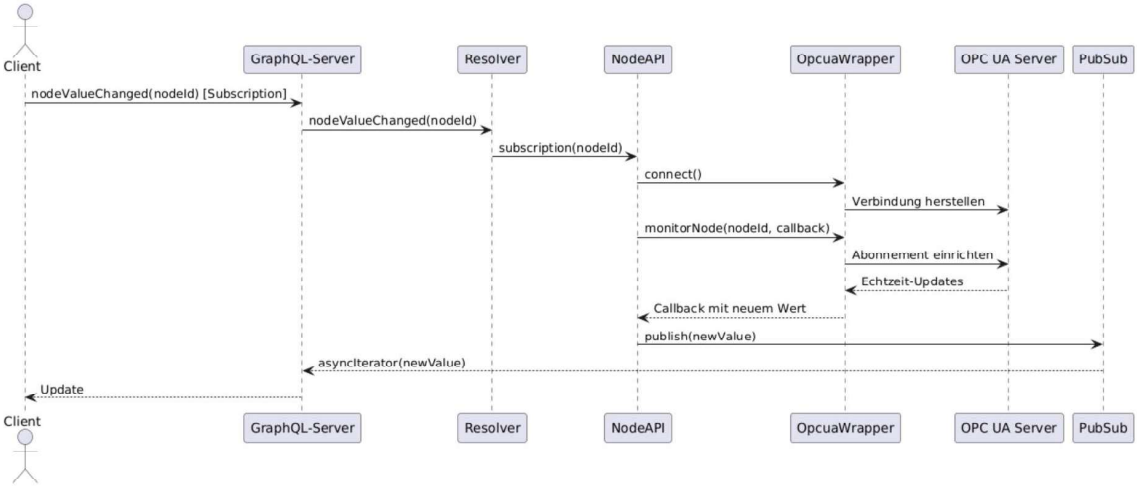


Abbildung 17: Sequenzdiagramm: Echtzeitüberwachung eines Knotens (nodeValueChanged) (Quelle: eigene Darstellung)

5.1.2.1 Schema.js

Es ist wichtig die Logik des OPC UA Address Space genau zu verstehen, bevor das GraphQL-Schema zu erstellen. Der Address Space kann als ein gerichteter Graph betrachtet werden, der aus mehreren Knoten besteht. Jeder Knoten besitzt eine eindeutige Identität (NodeId) und einen bestimmten Datentyp oder eine bestimmte Rolle im Modell. Diese Knoten sind miteinander über gerichtete Verbindungen (Referenzen), verbunden.

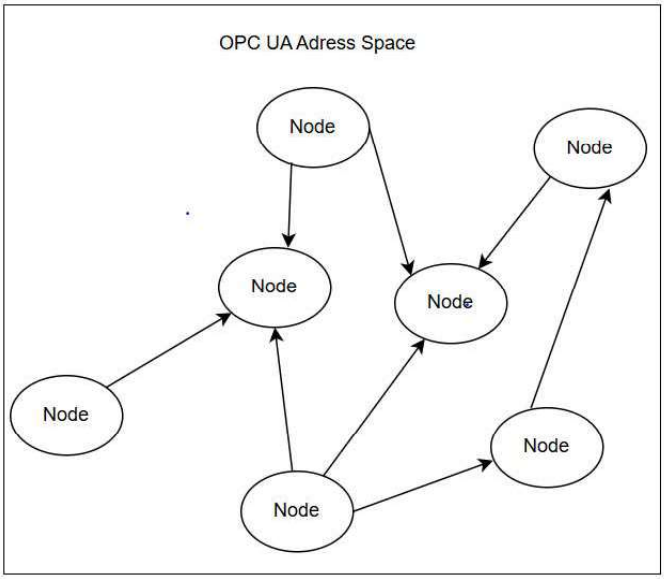


Abbildung 18: Beispiel von Adress Space OPC UA (Quelle: eigene Darstellung)

Das GraphQL-Schema wird in die JavaScript Datei „Schema.js“ erstellt und definiert eine Datenstruktur für die Knoten Typen und die Operationen auf dem OPC UA Adressraum nähmlich die Queries und Subscription Typen (Anhang 1).

Der `Typ Node` repräsentiert einen Knoten und modelliert die Eigenschaften eines Knotens in OPC UA Adressraum. Diese Struktur ermöglicht es, die hierarchische und referenzbasierte Natur von OPC UA-Modellen abzubilden. Ein Node enthält die eindeutige Identifikation `nodeId(String!)`, eine optionale Information zum Referenztyp `referenceType(String)`, ein `isForward(Boolean)` zur Richtung der Referenz sowie eine Liste von `children([Node!]!)`, welche die Kindknoten des aktuellen Knotens darstellen. Zusätzlich wird ein optionaler `value(String)` bereitgestellt, um den aktuellen Wert des Knotens zu speichern. Das `isVariabelTyp(Boolean)` zeigt an, ob es sich um einen Variablenknoten handelt. Diese Struktur ist entscheidend für das Modellieren und Visualisieren des Adressraum. Durch die Rekursives Natur des children-Feldes können Komplexe hierarchische Beziehungen im OPC UA-Modell effizient abgebildet und aufgerufen werden.

Der `Typ Query` definiert die Abfrage `getAllNodes (nodeId: String! maxEbene: Int)`. Mit dieser Abfrage lassen sich alle Knoten ab einem bestimmten Einstiegspunkt (nodeId) rekursiv abrufen. Die maximale Tiefe (maxEbene) kann hier optional begrenzt werden. Dies hilft, Daten effizient und ressourcenschonend zu laden. Mit `nodeValueChanged(nodeId: String!):Node` bietet das Schema eine Subscription um Änderungen eines Knotenwerts in Echtzeit zu überwachen

5.1.2.2 opcuaWrapper.js

Die Klasse opcuaWrapper abstrahiert die Kommunikation mit einem OPC UA-Server und bietet grundlegende Funktionen zum Verbindungsaufbau, Datenzugriff und Monitoring von Knoten. Ein OPC UA Client wird im Konstruktor initialisiert. Wichtige Parameter wie die Server-URL und Session-Informationen werden auch im Konstruktor verwaltet. Die Methode `connect ()` stellt eine Verbindung zum Server her und öffnet eine Sitzung. Die Methode `close ()` schließt die Sitzung und trennt die Verbindung. Mit `fetchNode(nodeId)` können Informationen zu einem bestimmten Knoten im Adressraum in Vorwärtsrichtung abgerufen werden (Anhang 3). Die Methode „`isVariableType(nodeId)`“ prüft, ob der Knoten zur Typ Variabel gehört. Die Methode `monitorNode(nodeId, newvalue)` überwacht Änderungen eines Knotens in Echtzeit. Dabei wird ein Abonnement eingerichtet und bei Wertänderungen die Callback-Funktion `newvalue` aufgerufen. Diese Struktur bietet eine klare und effiziente Schnittstelle für die clientseitige Interaktion mit dem OPC UA-Modell.

5.1.2.3 NodeAPI.js

Die Klasse nodeAPI ist die Schnittstelle zur Abfrage und Überwachung von Knoten im OPC UA Adressraum. Sie kombiniert die Funktionen zur rekursiven Datenerfassung und auch Echtzeit-Kommunikation über GraphQL-Subscriptions.
Im Konstruktor werden drei Komponenten initialisiert: eine Instanz von `OPCUAWrapper` zur Interaktion mit dem OPC UA-Server, eine Instanz von `PubSub` zur verwaltung von Subscriptions und `NODE_VALUE_CHANGED` für Knotenwert-Änderungen.
Die Methode `fetchNodesRecursively(nodeId, AllNode, maxEbene, visited)` ruft alle Knoten rekursiv ab (Anhang 4). Sie beginnt mit dem angegebenen Einstiegspunkt `nodeId`. Dabei kann optional eine maximale Rekursionstiefe durch `maxEbene` definiert werden. Besuchte Knoten werden mithilfe einer Set-Struktur `(visited)` markiert, um sicherzustellen, dass Knoten nicht mehrfach besucht werden. Mit `getAllNodes(nodeId, maxEbene)` wird die gesamte Knotenstruktur abgerufen. Diese Methode baut eine Verbindung zum Server auf, startet das rekursive Laden und

gibt schließlich alle gesammelten Knoten bis zur definierten Ebene zurück. Nach Abschluss wird die Verbindung zum Server geschlossen (Anhang 5).
Die Methode `subscription(nodeId)` richtet eine Echtzeitüberwachung für einen spezifischen Knoten ein. Änderungen am Knotenwert werden durch die Methode `monitorNode()` erfasst und über `pubsub.publish()` an die verbundenen Clients weitergegeben. Der Client erhaltet die neuen Werte über die vom GraphQL-Subscription Bibliothek predefiniertes Konzept „`asyncIterableIterator`“ (Anhang 6) .

5.1.2.4 Resolver.js

Die Datei „Resolver.js“ spielt eine zentrale Rolle in der API-Logik und sorgt für die Verarbeitung von GraphQL-Operationen durch Kommunikation mit dem OPC UA-Server (Anhang 7). Diese Datei ist in zwei Hauptbereiche unterteilt: Abfragen (Queries) und Subscriptions.
Der Resolver „`getAllNodes`“ befindet sich im Bereich der Abfrage. Er nimmt zwei Parameter, „`nodeId`“ und „`maxEbene`“, und stellt eine Verbindung zum OPC UA-Server her. Danach ruft er die die Methode „`getAllNodes`“ der „`NodeAPI`“-Instanz auf, um die Knoten im OPC UA-Baum entsprechend den angegebenen Kriterien abzurufen.
Der Resolver „`nodeValueChanged`“ befindet sich in den Bereich Subscriptions.
Er ruft die Methode „`Suscription`“ der „`NodeAPI`“ um die Überwachung der angegebenen Knoten einzurichten.

5.1.2.5 Index.js

Die index.js-Datei in der GraphQL-API spielt eine wichtige Rolle in der Struktur und Funktionalität der Anwendung. Sie konfiguriert den Server und stellt sowohl HTTP- als auch WebSocket-Endpunkte für Abfragen und Subscriptions bereit.
Der GraphQL-Server basiert auf dem Express-Framework und dem Apollo Server. Zur Integration von Subscriptions wird ein WebSocket-Server eingebunden.
In der Implementierung wird das Schema mithilfe von „`makeExecutableSchema`“ erstellt und der Resolver definiert. Anschließend erfolgt die Initialisierung eines HTTP-Servers durch die Verwendung von „`createServer()`“. Dieser ist in das Express-Framework integriert. Für die Kommunikation wird WebSocket-Server konfiguriert. „WebSocketServer“ wird eingesetzt und mit „`useServer`“ aus dem Paket „`graphql-ws`“ verbunden. Der Apollo Server wird mit dem definierten Schema initialisiert. Das Plugin „`ApolloServerPluginDrainHttpServer`“ sorgt hier für eine ordnungsgemäße Freigabe der Ressourcen beim Herunterfahren des Servers.

Schließlich wird der Server gestartet, wodurch Endpunkte für HTTP-Anfragen sowie für WebSocket-Verbindungen bereitgestellt werden (Anhang 8) [38].

5.2 Frontend (Visualisierung anhand einer Webanwendung)

Das Frontend ermöglicht die Visualisierung des OPC UA Address Space in Form eines interaktiven Graphen. Die Anwendung soll es den Benutzern ermöglichen, die Struktur der Knoten zu erkunden, indem sie eine Start-Knoten-ID eingeben und die Tiefe des Baums bestimmen. Dies unterstützt die Benutzer dabei, spezifische Informationen schnell zu finden und zu analysieren.
Das Frontend wird mit dem Framework Angular implementiert.

5.2.1 Webanwendung mit Angular

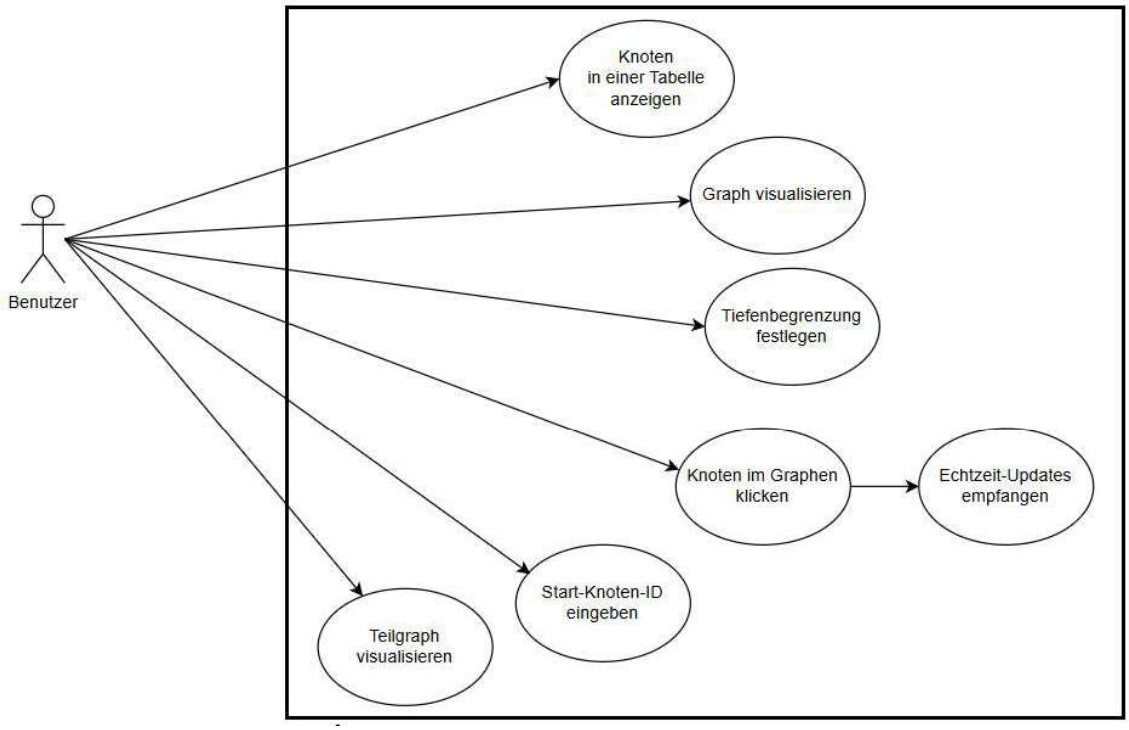


Abbildung 19: Use-Case-Diagramm Benutzeroberfläche (Quelle: eigene Darstellung)

Um diese Benutzer Oberfläche zu erstellen, müssen folgende Anforderungen erfüllt werden (Anhang 16).

- FA-01: Darstellung des Adressraums (Hoch).
- FA-02: Echtzeit-Aktualisierung (Hoch).
- FA-03: Filterung (Mittel).
- FA-04: Caching (Hoch).
- FA-05: Antwortzeiten (Mittel).

Die Datei „app.module.ts“ bildet das Fundament der Angular-Anwendung. Sie definiert die Struktur und die Abhängigkeiten des Moduls. Sie organisiert Komponenten, integriert Dienste und legt der Startkomponente fest. Dies ist, für die korrekte Funktion der Anwendung, entscheidend [39].

In dieser Datei wurde eine Caching-Strategie mit `InMemoryCache` implementiert (Anhang 9). Dies verbessert die Performance der Anwendung.

Zusätzlich wurde WebSocket-Subscription-Unterstützung `(GraphQLWsLink)` integriert, um Echtzeit-Updates zu ermöglichen.

Die Anwendung besteht aus zwei Hauptkomponenten: „Home“ und „Graph“. In der „Home“ Komponente werden alle Konten und deren Referenzen in einer übersichtlichen Tabelle angezeigt. Diese Tabelle ermöglicht es den Benutzern, schnell einen Überblick über die verfügbaren Knoten zu erhalten. Die „Graph“-Komponente hingegen ermöglicht die grafische Darstellung des AddressSpaces. Die Knoten des Address Spaces werden als separate Elemente dargestellt, während ihre Beziehungen und Hierarchien durch Kanten visualisiert werden. Hier können Benutzer die eingegebene Knoten-ID visualisieren und die Beziehungen zwischen den Knoten durch die Interaktion mit dem Graphen erkunden.

Benutzer können die Start-Knoten-ID eingeben, die entweder die RootFolder oder eine andere spezifische Node-ID sein kann. Zusätzlich haben sie die Möglichkeit, die Tiefe des angezeigten Baums festzulegen, wobei sie entweder die gesamte Struktur oder eine bestimmte Tiefe angeben können (Anhang 10).

In der Graph-Komponente wird eine Farbmarkierung für verschiedene Knotenklassen eingeführt. Variablenknoten `(nodeClass: Variable)` werden rot dargestellt, während alle anderen Knoten blau sind.

Zudem wird eine WebSocket-Subscription `(nodeValueChanged)` implementiert. Wenn ein Benutzer auf einen Variablen-Knoten klickt, wird dessen aktueller Wert in Echtzeit abgerufen und angezeigt.

Es wird auch die `Fetch-Policy cache-first` genutzt, um zuerst Daten aus dem Cache zu laden, bevor eine neue Anfrage gesendet wird. Dadurch wird die Performance deutlich verbessert.

Die Visualisierung des AddressSpaces erfolgt mithilfe der VisNetwork-Bibliothek. Diese Bibliothek bietet eine flexible und leistungsstarke Möglichkeit, Daten in einem interaktiven Graphen darzustellen. Die Knoten und Kanten des Graphen werden entsprechend den Adressdaten angelegt, wobei verschiedene Eigenschaften wie Knotenfarben und -größen verwendet werden können, um unterschiedliche Informationen hervorzuheben. Darüber hinaus unterstützt VisNetwork Animationen und

Interaktionen, die es den Benutzern ermöglichen, dynamisch durch den Graphen zu navigieren und tiefere Einblicke in die Struktur der Daten zu erhalten.

5.2.2 Kommunikation zwischen Angular und GraphQL

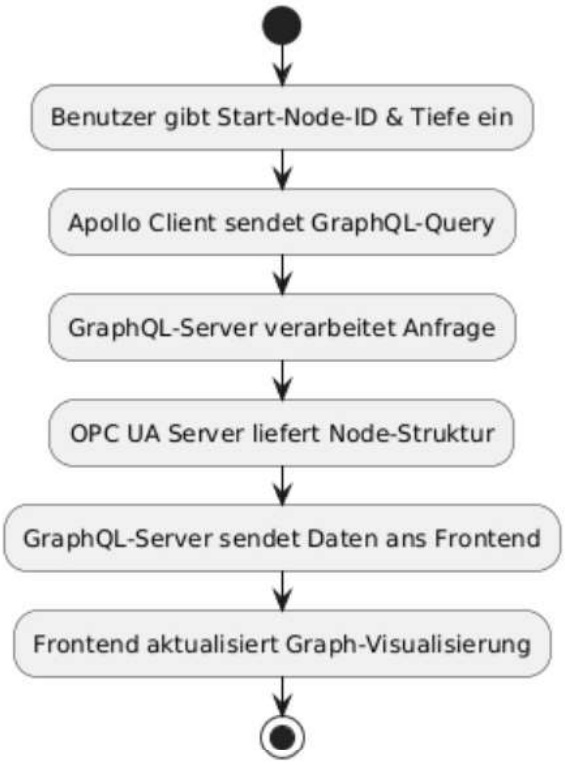


Abbildung 20: Datenflussdiagramm der Anwendung (Quelle: eigene Darstellung)

Apollo Client ermöglicht die Kommunikation zwischen Angular und der GraphQL-API und ist in der Hauptmodul Datei „app.module.ts“ konfiguriert. Hier ist die URL der GraphQL-API definiert und ein HTTP-Link eingerichtet. Um Daten abzurufen, werden in den Angular-Komponenten spezifische GraphQL-Abfragen formuliert.
Die Anwendung verwendet GraphQL als Schnittstelle zwischen dem Frontend und dem OPC UA-Server. Der Datenfluss zwischen Frontend und Backend erfolgt über ein asynchrones Kommunikationsmodell. Wenn ein Benutzer beispielsweise in der Graph-Komponente eine Start-Knoten-ID eingibt und die Tiefe des Baums bestimmt, wird eine GraphQL-Query generiert, die diese Parameter berücksichtigt. Diese Anfrage wird dann an den GraphQL-Server gesendet, der die entsprechenden Daten aus dem OPC UA-Server abruft.
Nach der Verarbeitung der Anfrage sendet das Backend die angeforderten Daten in einem strukturierten Format zurück an das Frontend. Die Frontend-Anwendung verarbeitet diese Daten und aktualisiert die Benutzeroberfläche in den Graphe Komponente entsprechend.

6 Evaluation

Das Ziel dieser Bachelorarbeit war, den Zugriff auf das OPC UA-Datenmodell zu optimieren und sowohl die Backend- als auch die Frontend-Performance zu verbessern. Ausgangspunkt war ein Webanwendung, die über eine REST-API mit einem OPC UA-Server kommunizierte, die aufgrund ihrer Architektur erhebliche Performance-Probleme aufwies. Um die Effizienz zu steigern und die Benutzererfahrung zu verbessern, wurden mehrere Optimierungen implementiert. In diesem Abschnitt werden die wichtigsten Verbesserungen erläutert, die durch die Entwicklung einer GraphQL-API und die Anpassung des Frontend erreicht wurden.
Diese Untersuchung vergleicht zwei verschiedene Ansätze zur Visualisierung des OPC UA Adressraums in einer Webanwendung:

- Ansatz 1 (Ausgangspunkt): Ein Frontend, das über eine REST-API mit dem OPC UA-Server kommuniziert.
- Ansatz 2 (unsere Lösung): Ein Frontend, das über eine GraphQL-API mit dem OPC UA-Server interagiert.

Die Ansätze unterscheiden sich in der Art der Kommunikation mit dem OPC UA-Server, insbesondere bei der Abfrage der Knoten und Referenzen.

6.1 Technische Analyse und Bewertung der beiden Ansätze

Die folgende Tabelle zeigt die beobachteten Antwortzeiten.

Tabelle 7: Vergleich der Antwortzeiten zwischen REST-API und GraphQL-API (mit/ohne Caching) (Quelle: eigene Darstellung)

Methode	Antwortzeit ohne Caching	Antwortzeit mit Caching
REST-API	mehrere Stunden	-
GraphQL-API	20 Sekunden	2 Sekunden

6.1.1 Ansatz mit REST-API

6.1.1.1 Funktionsweise

Bei der REST-API wird ein sequentieller Ansatz gefolgt. Für jede Knotenabfrage wird eine neue Verbindung zum OPC UA-Server aufgebaut. Eine separate Session wird für jede Knoten erstellt, um die Referenzen des Knotens abzurufen. Nach Abschluss der Anfrage wird die Session geschlossen und die Verbindung getrennt. Dieses Vorgehen führt zu einem erheblichen Overhead.

6.1.1.2 Leistungsprobleme

Die wiederholte Erstellung und Beendigung von Sessions verursacht erhebliche Verzögerungen, da jeder Verbindungsaufbau Zeit und Ressourcen benötigt. Netzwerk- und Verbindungsaufbauzeiten führen dazu, dass die Gesamtantwortzeit mehrere Stunden beträgt. Zudem fehlt eine effiziente Möglichkeit, mehrere Knoten gleichzeitig abzurufen. Ein weiteres Defizit dieses Ansatzes ist die fehlende Echtzeitfähigkeit, da Änderungen im Adressraum nicht unmittelbar erkannt werden können.
Bei der REST-API beträgt die durchschnittliche Zeit pro Knoten etwa 100 Millisekunden. Dies ist die Zeit, die erforderlich ist, um eine Antwort für einen einzelnen Knoten zu erhalten, was für 5300 Knoten 530 Sekunden entspricht.

6.1.2 Ansatz mit GraphQL-API

6.1.2.1 Funktionsweise

Im Gegensatz zur REST-API nutzt die GraphQL-API eine einzige dauerhafte Session, um den gesamten Adressraum in einer einzigen Abfrage zu erfassen. Die Notwendigkeit, für jede Knotenabfrage eine neue Verbindung herzustellen, entfällt. Zusätzlich ermöglicht GraphQL die Nutzung von Subscriptions, um Änderungen am Adressraum in Echtzeit an das Frontend übermittelt werden. Zudem wurde ein Caching-Mechanismus Clientseite implementiert.

6.1.2.2 Verbesserung durch den Einsatz einer einzigen Session

Reduktion der Anfragen: Der gesamte Adressraum wird in einer einzigen GraphQL-Query abgerufen.
Effizienzsteigerung: Die Antwortzeit verringert sich von mehreren Stunden auf 20 Sekunden (ohne Caching).
Geringere Netzwerkbelastung: Reduzierung des Netzwerklastes durch die optimierte Abfrage (weniger einzelne Requests werden gesendet).

6.1.2.3 Verbesserung durch den Einsatz von Subscriptions für Echtzeit-Updates

Ein weiterer interessanter Vorteil von GraphQL gegenüber REST ist die native Unterstützung von Subscriptions. Wenn der Wert eines Knotens sich ändert, wird die Information automatisch an das Frontend übertragen, ohne dass ein periodisches

Polling erforderlich ist. Dies reduziert den Netzwerkverkehr und verbessert die Reaktionsfähigkeit des Systems.

6.1.2.4 Verbesserung durch den Einsatz von Caching-Mechanismus im Frontend

Um die Leistung weiter zu optimieren, wurde ein Caching-Mechanismus implementiert. Tests zeigen, dass eine Abfrage ohne Caching rund 20,5 Sekunden für 5300 Knoten benötigt, während dieselbe Abfrage mit Caching nur 2,1 Sekunden dauert. Dies entspricht einer Reduzierung der Ladezeit um etwa 92,8 %.

6.2 Zusammenfassung

Diese Analyse zeigt in unserem speziellen Anwendungsfall, dass der Einsatz der GraphQL-API zusammen mit einer einzigen Sitzung zur rekursiven Abfrage der Daten sowie der zusätzlichen Nutzung von Caching im Frontend erheblich effektiver ist als die ursprüngliche REST-API. Sie bietet eine drastische Reduktion der Antwortzeiten, höhere Effizienz, bessere Skalierbarkeit und eine Echtzeitüberwachung des Adressraums.

7 Fazit und Ausblick

Im Rahmen dieser Bachelorarbeit wurden verschiedene Optimierungensmaßnahmen zur Verbesserung der bestehenden WebApp-REST-API in Verbindung mit einem OPC UA-Server untersucht und implementiert. Die Einführung einer einzigen Session, rekursive Datenverarbeitung sowie gezielten Datenabruf mittels GraphQL konnten signifikante Leistungsverbesserungen erzielen. Die Unterstützung von Subscriptions im Backend und das gezielte Filtern nach Knoten im Frontend sowie die Nutzung von Caching haben zusätzlich die Benutzererfahrung verbessert.

Diese Änderungen haben einerseits die Performance der Anwendung erhöht und andererseits eine intuitivere und flexiblere Interaktion mit dem komplexen OPC UA-Datenmodell ermöglicht.

Die Wahl von GraphQL in Kombination mit den implementierten Techniken stellet eine zukunftssichere Lösung für den Zugriff auf komplexe Datenmodelle dar. Der entwickelte Prototyp hat die Machbarkeit der vorgeschlagenen Optimierungen demonstriert, auch wenn die zeitlichen Einschränkungen einer Bachelorarbeit die Möglichkeit einer umfassenden Implementierung und Validierung in einer realen industriellen Umgebung begrenzt haben.

Für zukünftige Arbeiten und Forschungen wäre die Implementierung von GraphQL in einer echten industriellen Umgebung interessant zu untersuchen. Es wäre von Interesse, zu analysieren, wie die API unter der Belastung mehrerer Clients reagiert und ob die Performancevorteile in einer produktiven Umgebung auch tatsächlich realisiert werden können.

Zusätzlich könnte untersucht werden, ob die gleichen Optimierungen auch für eine REST-API implementiert und anschließend mit der entwickelten GraphQL-API in einem industriellen Umfeld verglichen werden können.

Insgesamt eröffnen die durchgeführten Optimierungen und deren Ergebnisse zahlreiche Ansätze für zukünftige Entwicklungen, die sowohl die Leistungsfähigkeit als auch die Benutzerfreundlichkeit von APIs in industriellen Anwendungen weiter steigern können.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, vorliegende Arbeit selbstständig und ohne Zuhilfenahme unzulässiger Hilfsmittel angefertigt zu haben. Wörtliche oder dem Sinne nach übernommenen Ausführungen sind gekennzeichnet, sodass Missverständnisse über die geistige Urheberschaft ausgeschlossen sind. Diese Arbeit war bisher noch nicht Bestandteil einer Studien- oder Prüfungsleistung in gleicher oder ähnlicher Fassung.

Ort, den 6. März 2025

Anhangsverzeichnis

Anhang

Anhang 1: Codeausschnitt: GraphQL-Schema für OPC UA-Knoten mit Abfragen und Subscriptions (eigene Darstellung)

```
const typeDefs : DocumentNode  = gql`
  type Node {
    nodeId: String!
    referenceType: String
    isForward: Boolean
    children: [Node!]!
    value:String
    isVariabelTyp: Boolean
  }
  type Query {
    getAllNodes (nodeId: String!,maxEbene: Int): [Node!]!
  }
  type Subscription {
  nodeValueChanged(nodeId:String!):Node
  }
```

Anhang 2: Abfrage aller OPC UA-Knoten mit optionaler Tiefenbeschränkung (eigene Darstellung)

```
query GetAllNodes {
  getAllNodes(nodeId:"ns=0;i=84",maxEbene:null)
  {
      nodeId
      isVariabelTyp
      children {
          nodeId
      }
  }
}

```

Anhang 3: Codeausschnitt (fetchNode-Funktion): Durchsuchen eines OPC UA-Knotens mit Fehlerbehandlung (eigene Darstellung)

```
async fetchNode(nodeId) : Promise<...> {
    try{
        return await this.session.browse({
            nodeId: nodeId,
            browseDirection: BrowseDirection.Forward,
            resultMask: 0b111111,
        });
}catch (error){
        console.error("fehler beim Browsen",error.message);
      return null;
    }
}
```

Anhang 4: Codeausschnitt: Rekursives Durchsuchen von OPC UA-Knoten und Hierarchieaufbau (eigene Darstellung)

```
async  fetchNodesRecursively( nodeId,AllNode, maxEbene, visited : Set<any>  = new Set()) : Promise<...>
{
    if (visited.has(nodeId)) {
        const childnode = AllNode.find(node => node.nodeId === nodeId);
        return childnode;}
    visited.add(nodeId);
    const browseResult : any | null | undefined  = await this.opcua.fetchNode(nodeId);
    const AktuelleKnote : {children: ..., nodeId: string}  = {
        nodeId: nodeId.toString(),
        children: [],};
    AktuelleKnote.isVariabelTyp = await
        this.opcua.isVariableType(AktuelleKnote.nodeId);
    if (maxEbene!==null){
        maxEbene = maxEbene-1;}
    for (const reference of browseResult.references) {
        const childId : string  = reference.nodeId.toString();
        if (maxEbene === null || maxEbene > 0) {
            const childNode = await this.fetchNodesRecursively(childId, AllNode, maxEbene,visited);
            if (childNode) {
                AktuelleKnote.children.push(childNode);
            } } }
    AllNode.push(AktuelleKnote);
    return AktuelleKnote;}
```

Anhang 5: Codeausschnitt: Funktion zum Laden aller OPC UA-Knoten (eigene Darstellung)

```
async getAllNodes(nodeId,maxEbene) : Promise<...>
{
    try {
        // Verbindung zum OPC UA-Server aufbauen
        await this.opcua.connect();
        let AllNode : any[] =[];
        console.log("Rekursives Laden für NodeId:"+ nodeId +"gestartet...");
        await this.fetchNodesRecursively( nodeId,AllNode,maxEbene);
        //console.log("Alle Knoten erfolgreich geladen:", AllNode);
        return AllNode;
    } catch (error) {
        console.error("Fehler im Resolver getAllNodes:", error.message);
        throw new Error("Fehler beim Abrufen der Knoten.");
    }
    finally {
        await this.opcua.close();
    }
}
```

Anhang 6: Codeausschnitt Subscription für OPC UA-Knotenwertänderungen mit PubSub-Mechanismus (eigene Darstellung)

```
async subscription (nodeId) : Promise<...> {
    try{
        await this.opcua.connect()
        this.opcua.monitorNode(nodeId, newvalue: (newValue) : void => {
            this.pubsub.publish(this.NODE_VALUE_CHANGED, payload: {
                nodeValueChanged :{
                    nodeId,
                    value:newValue,
                },
            });
        });
        return this.pubsub.asyncIterableIterator(this.NODE_VALUE_CHANGED);
    }catch (error){
        console.error("Fehler im Subscription",error.message);
    }
}
```

Anhang 7: Codeausschnitt: Resolvers für OPC UA-Baumdurchlauf und Echtzeit (eigene Darstellung)

```
const resolvers : {Query: {...}, Subscription: {...}}  = {
    Query: {
        // Query zur Rekursion durch den OPC UA-Baum
        getAllNodes: async (_, {nodeId, maxEbene}) : Promise<...>  => {
            const endpointUrl : string  = "opc.tcp://localhost:53530/OPCUA/SimulationServer";
            const nodeAPI : nodeAPI  = new NodeAPI(endpointUrl);
            return await nodeAPI.getAllNodes(nodeId, maxEbene);
        },
    },
    Subscription:{
    nodeValueChanged:{
        subscribe:async (_,{nodeId}) : Promise<...> =>{
            const endpointUrl : string  = "opc.tcp://localhost:53530/OPCUA/SimulationServer";
            const nodeAPI : nodeAPI  = new NodeAPI(endpointUrl);
            return await nodeAPI.subscription(nodeId);
        }
    }
}
};
```

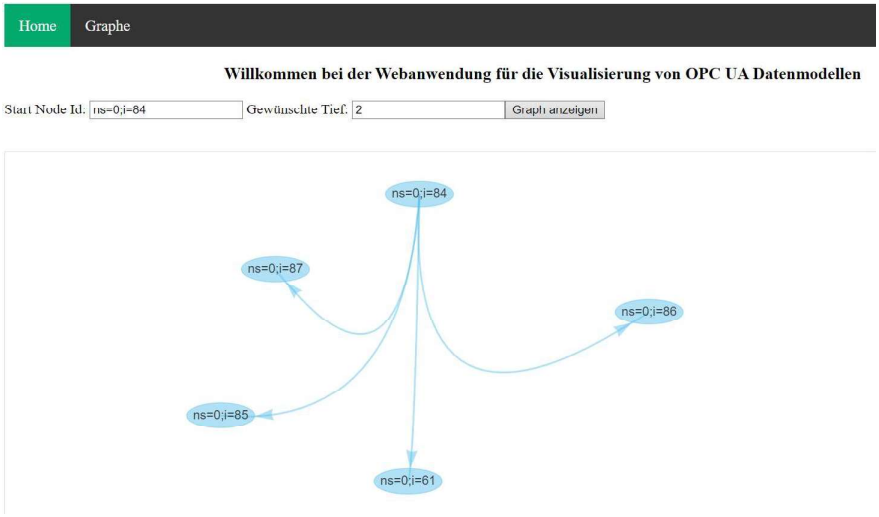
Anhang 8: Codeausschnitt: GraphQL-Server mit Apollo Server und Subscriptions über WebSocket (eigene Darstellung)

```
async function startServer() : Promise<void>  {
  await server.start();
  server.applyMiddleware( {app, ...rest}: { app });
  httpServer.listen( port: 4000,  hostname: () : void  => {
    console.log("Query endpoint ready at http://localhost:4000/graphql");
    console.log("Subscription endpoint ready at ws://localhost:4000/graphql");
  });
}
```

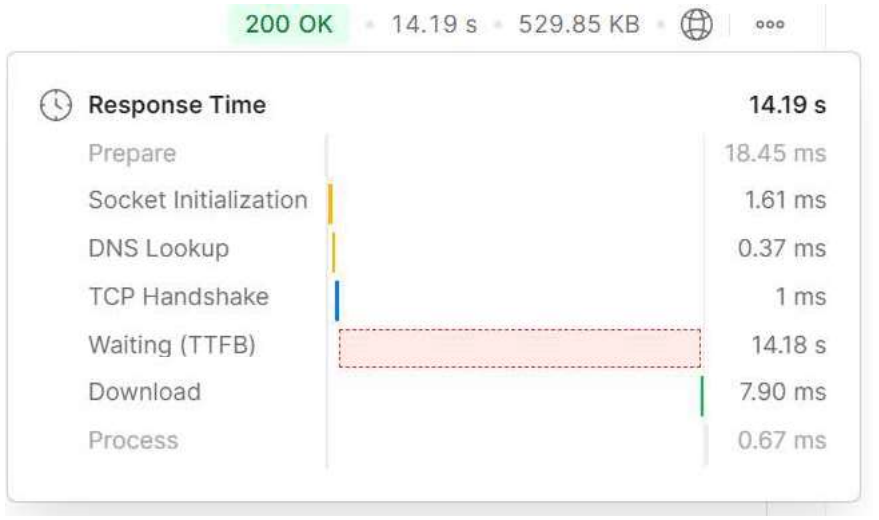
Anhang 9: Codeausschnitt: Cache-Konfiguration mit nodeId als Schlüsselfeld in Apollo Client (eigene Darstellung)

```
cache: new InMemoryCache( config: {
  typePolicies: {
    Node: {
      keyFields: ['nodeId'], // Verwende 'nodeId' als eindeutige ID
    },
  },
})
```

Anhang 10: Webanwendung zur Visualisierung von OPC UA (eigene Darstellung)



Anhang 11: Ladezeit für den Abruf aller Knoten in GraphQL (eigene Darstellung)



Anhang 12: Antwortzeit ohne Caching im Frontend (GraphQL-API) (eigene Darstellung)

```
Gesamte-Ladezeit: 20560.126953125 ms                    graph.component.ts:77
```

Anhang 13: Antwortzeit mit Caching im Frontend (GraphQL-API) (eigene Darstellung)

```
Gesamte-Ladezeit: 2126.421142578125 ms                  graph.component.ts:77
```

Anhang 14: Funktionale Anforderungen (GraphQL-API) (Quelle: eigene Darstellung)

Funktionale Anforderungen (FA)		
Nr./Id	FA-01	**Titel:**
Prioritäten	Hoch	Zugriff auf OPC UA Knoten und deren Hierarchie
Beschreibung: alle Knoten und deren Hierarchien in einer einzigen Abfrage erhalten.		
Nr./Id	FA-02	**Titel:**
Prioritäten	Hoch	Verbindung zum OPC UA-Server
Beschreibung: Herstellung und Verwaltung von Verbindungen zu einem OPC UA-Server. (Verbindung, Sitzung, Ausführung von Abfragen, Fehlerbehandlung)		
Nr./Id	FA-03	**Titel:**
Prioritäten	Mittel	Datenstruktur für Knoten
Beschreibung: Unterstützung verschiedener Datentypen im OPC UA-Modell und deren Bereitstellung in GraphQL Schema.		
Nr./Id	FA-04	**Titel:**
Prioritäten	Mittel	Filterung der Knoten bis zu einer maximalen Tiefe
Beschreibung: Die maximale Tiefe der Knotenstruktur zu definieren und bis zur angegebenen Ebene die Ergebnisse zurückgeben.		
Nr./Id	FA-05	**Titel:**
Prioritäten	Mittel	Abfrage aller Knoten eines bestimmten Startpunktes.
Beschreibung: Die Gesamte Knotenstruktur ausgehend von einer beliebigen „nodeId“ durchlaufen.		
Nr./Id	FA-06	**Titel:**
Prioritäten	Hoch	Echtzeit Benachrichtigung für Knotenwertänderungen
Beschreibung: Bereitstellung von Echtzeit-Benachrichtigungen bei Änderungen der Knotenwert.		
Nr./Id	FA-07	**Titel:**
Prioritäten	Hoch	Leistung
Beschreibung: Die Anfragen müssen in einem angemessenen Zeitrahmen beantwortet werden, insbesondere bei großen Baumstrukturen.		

Anhang 15: Nicht Funktionale Anforderungen (GraphQL-API)

(Quelle: eigene Darstellung)

Nicht Funktionale Anforderungen (NFA)		
Nr./Id	NFA-01	**Titel:**
Prioritäten	Hoch	Code-Qualität
Beschreibung: Der Code soll auf einer verbreiteten Programmiersprache geschrieben werden. Um spätere Anpassungen durch andere Personen problemlos zu ermöglichen, muss der Code leicht zu verstehen sein		
Nr./Id	NFA-02	**Titel:** Standards & Frameworks
Prioritäten	Hoch	
Beschreibung: Bekannte und erprobte Standards, SDKs und Frameworks sollten verwendet werden.		
Nr./Id	NFA-03	**Titel:**
Prioritäten	mittel	Dokumentation
Beschreibung: Die API sollte gute Dokumentation bereitstellen, die für Entwicklern leicht verständlich ist.		
Nr./Id	NFA-04	**Titel:** Lizenzfreie Software
Prioritäten	Hoch	
Beschreibung: Die Software soll keine kosten- pflichtigen Plug-Ins, SDKs oder Frameworks nutzen.		

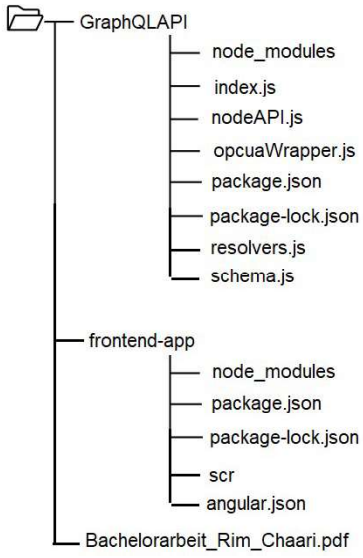
Anhang 16: Funktionale Anforderungen (Webanwendung) (Quelle: eigene Darstellung)

Funktionale Anforderungen (FA)		
Nr./Id	FA-01	**Titel:**
Prioritäten	Hoch	Darstellung des Adressraums
Beschreibung: Alle Knoten und Verbindungen des OPC UA-Servers sollen visualisiert werden.		
Nr./Id	FA-02	**Titel:**
Prioritäten	Hoch	Echtzeit-Aktualisierung von Knotenwerten
Beschreibung: Wenn ein Benutzer auf einen Knoten klickt und dessen Wert sich ändert, soll der neue Wert in Echtzeit visualisiert werden.		
Nr./Id	FA-03	**Titel:**
Prioritäten	Mittel	Filterung des Adressraums
Beschreibung: Möglichkeit, nur bestimmte Knoten und deren Verbindungen darzustellen.		
Nr./Id	FA-04	**Titel:**
Prioritäten	Hoch	Caching-Mechanismus
Beschreibung: Optimierung der Performance durch Caching von häufig abgerufenen Daten.		
Nr./Id	FA-05	**Titel:**
Prioritäten	Mittel	Schnelle Antwortzeiten
Beschreibung: Minimierung der Latenz bei Abfragen des Adressraums.		

Anhang 17: Zusätzliche Materialien

Eine CD mit den folgenden Inhalten liegt dieser Arbeit bei:

- Quellcode der GraphQL-API.
- Quellcode der Frontend.
- Anfertigung der Bachelorarbeit



Zusätzlich befindet sich der Gesamtcode unter folgen GitHub Links:

https://github.com/RimCh23/GraphQL_API_Rekursiv.git

https://github.com/RimCh23/GraphQL_Frontend.git

Litteraturverzeichnis

[1] Wikipedia. (2025). OPC Unified Architecture. Verfügbar unter: https://de.wikipedia.org/wiki/OPC_Unified_Architecture. Abgerufen am: [04.02.25].

[2] OPC Foundation. (2017). OPC UA Interoperability für Industrie 4.0 und IoT. Verfügbar unter: https://opcfoundation.org/wp-content/uploads/2017/11/OPC-UA-Interoperability-For-Industrie4-and-IoT-DE.pdf. Abgerufen am: [04.02.25].

[3] M. Schleipen, S. Faul, und A. Wiesner, "REST-based OPC UA for the Industrial Internet of Things (IIoT)," 2018. [Online]. Verfügbar unter: https://www.ias.uni-stuttgart.de/dokumente/publikationen/2018_rest_based_opc_ua_for_the_iiot.pdf. Abgerufen am: [04.02.25].

[4] Siemens AG, "OPC UA mit SIMATIC S7-1500 – Kommunikation über OPC UA," 2018. [Online]. Verfügbar unter: https://www.automation.siemens.com/sce-static/learning-training-documents/tia-portal/advanced-communication/sce-092-300-opc-ua-s7-1500-r1807-de.pdf.

[5] Wikipedia, "Representational State Transfer." [Online]. Verfügbar unter: https://de.wikipedia.org/wiki/Representational_State_Transfer. Abgerufen am: [04.02.25].

[6] Apollo GraphQL, "Resolvers." [Online]. Verfügbar unter: https://www.apollographql.com/docs/apollo-server/data/resolvers. Abgerufen am: [04.02.25].

[7] R. T. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," Ph.D. Thesis, University of California, Irvine, CA, USA, 2000.

[8] OPC Foundation, "OPC UA Specification Part 3: Address Space Model (v1.04)." [Online]. Verfügbar unter: https://reference.opcfoundation.org/Core/Part3/v104/docs/4.3. Abgerufen am:04.02.25

[9] OPC Foundation, "6.3 Session-less Service Invocation," OPC Unified Architecture Part 4: Services, Version 1.04. [Online]. Verfügbar: https://reference.opcfoundation.org/Core/Part4/v104/docs/6.3. [Zugriff: 31.01. 2025].

[10] S. Gruner, J. Pfrommer und F. Palm, "RESTful Industrial Communication with OPC UA," IEEE Transactions on Industrial Informatics, vol. 12, pp. 1832–1841, 2016. [Online]. Verfügbar unter: https://www.researchgate.net/publication/294723351_RESTful_Industrial_Communication_with_OPC_UA.

[11] Cylynx.io: A Comparison of JavaScript Graph Network Visualisation Libraries. Zugriff am [31.01.2025]. Verfügbar unter: https://www.cylynx.io/blog/a-comparison-of-javascript-graph-network-visualisation-libraries/.

[12] GraphQL Foundation, "Introduction to GraphQL." [Online]. Verfügbar unter: https://graphql.org/learn/. Abgerufen am: [03.02.25].

[13] Wikipedia, "GraphQL." [Online]. Verfügbar unter: https://de.wikipedia.org/wiki/GraphQL. Abgerufen am: [03.02.25].

[14] GraphQL Foundation, "GraphQL." [Online]. Verfügbar unter: https://graphql.org/. Abgerufen am: [03.02.25].

[15] Hygraph, "Products using GraphQL: Netflix revolutionizing video content delivery with GraphQL." [Online]. Verfügbar unter: https://hygraph.com/blog/products-using-graphql#netflix-revolutionizing-video-content-delivery-with-graphql. Abgerufen am: [03.02.25].

[16] Apollo GraphQL, "Schema Basics." [Online]. Verfügbar unter: https://www.apollographql.com/docs/apollo-server/schema/schema/. Abgerufen am: [03.02.25].

[17] GitHub, "Introduction to GraphQL: Argument." [Online]. Verfügbar unter: https://docs.github.com/de/graphql/guides/introduction-to-graphql#argument. Abgerufen am: [03.02.25].

[18] GraphQL Foundation, "Response." [Online]. Verfügbar unter: https://graphql.org/learn/response/. Abgerufen am: [03.02.25].

[19] GraphQL Foundation, "GraphQL Specification: Data," October 2021. [Online]. Verfügbar unter: https://spec.graphql.org/October2021/#sec-Data. Abgerufen am: [03.02.25]

[20] GeeksforGeeks, "Resolvers in GraphQL." [Online]. Verfügbar unter: https://www.geeksforgeeks.org/resolvers-in-graphql/. Abgerufen am: [03.02.25].

[21] Wikipedia, "AngularJS." [Online]. Verfügbar unter: https://de.wikipedia.org/wiki/AngularJS. Abgerufen am: [04.02.25].

[22]R. J. Publications, IJCSP23A1361: Sicherheit in IoT- und Kommunikationssystemen, [Online]. Verfügbar: https://rjpn.org/ijcspub/papers/IJCSP23A1361.pdf. Zugriff am: 5. Februar 2025.

[23] BMU Verlag. (n.d.). Was macht ein Angular Entwickler und was braucht er dafür? Abgerufen am 05.02.25 von https://bmu-verlag.de/was-macht-ein-angular-entwickler-und-was-braucht-er-dafuer/

[24] entwickler.de, "Web App Tutorial mit Angular." [Online]. Verfügbar unter: https://entwickler.de/angular/web-app-tutorial-angular. Abgerufen am: [05.02.25].

[25] Angular, "Component Styles." [Online]. Verfügbar unter: https://v17.angular.io/guide/component-styles. Abgerufen am: [05.02.25].

[26] M. Alaskari, "Effektives Routing in Angular 17: Ein Leitfaden," Medium, [Online]. Verfügbar unter: https://medium.com/@mohamad-alaskari/effektives-routing-in-angular-17-eine-leitfaden-a2608d949a71. Abgerufen am: [05.02.25].

[27] eology, "Angular – Erklärung und Vorteile." [Online]. Verfügbar unter: https://www.eology.de/wiki/angular. Abgerufen am: [05.02.25].

[28] OPC Foundation, "The State of Things," OPC Connect, Juni 2017. [Online]. Verfügbar unter: https://opcconnect.opcfoundation.org/2017/06/the-state-of-things/. Abgerufen am: [05.02.25].

[29] GraphQL Foundation, "Subscriptions." [Online]. Verfügbar unter: https://graphql.org/learn/subscriptions/. Abgerufen am: [05.02.25]

[30] Kinsta, "GraphQL vs. REST: Was ist der Unterschied?" [Online]. Verfügbar unter: https://kinsta.com/de/blog/graphql-vs-rest/. Abgerufen am: [05.02.25].

[31] RestfulAPI.net, "Caching in RESTful APIs." [Online]. Verfügbar unter: https://restfulapi.net/caching/. Abgerufen am: [05.02.25].

[32] Apollo GraphQL, "Caching in Apollo Client." [Online]. Verfügbar unter: https://www.apollographql.com/docs/react/caching/overview. Abgerufen am: [05.02.25].

[33] Z. Zhou, L. Zhang, und M. Li, "Research on Data Sharing and Access Control Based on RESTful API in IoT Environment," Computers, vol. 11, no. 5, Art. 65, 2022. [Online]. Verfügbar unter: https://www.mdpi.com/2073-431X/11/5/65. Abgerufen am: [05.2.25].

[34] GraphQL .NET, "Using GraphiQL with GraphQL .NET." [Online]. Verfügbar unter: https://graphql-dotnet.github.io/docs/getting-started/graphiql/. Abgerufen am: [05.02.25].

[35] OPC Foundation, "OPC UA Part 1: Overview and Concepts," accessed [10.02.25], [Online]. Available: https://reference.opcfoundation.org/Core/Part1/v104/docs/3.43

[36] Z. Zhou, L. Zhang und M. Li, "Comparison of REST and GraphQL Interfaces for OPC UA," Computers, Bd. 11, Nr. 5, S. 65, 2022. DOI: https://doi.org/10.3390/computers11050065.

[37] D. Schmitt, "AngularJS und TypeScript," angular.de, [Online]. Verfügbar: https://angular.de/artikel/angularjs-und-typescript/. [Zugriff 4.02.25].

[38] Apollo GraphQL, "Subscriptions in Apollo Server." [Online]. Verfügbar unter: https://www.apollographql.com/docs/apollo-server/data/subscriptions. Abgerufen am: [04.02.25]

[39] Angular, "NgModules – Angular v17 Guide." [Online]. Verfügbar unter: https://v17.angular.io/guide/ngmodules. Abgerufen am: [04.02.25].

[40] OPC Router, "Was ist OPC UA?" [Online]. Verfügbar unter: https://www.opc-router.de/was-ist-opc-ua/. Abgerufen am: [10.02.25].

[41] N. Mühlbauer, E. Kirdan, M.-O. Pahl und G. Carle, "Open-Source OPC UA Security and Scalability," Technical University of Munich, Covalion, Framatome, IMT Atlantique.

[42] R. Schiekofer und M. Weyrich, "Introduction of Group-Subscriptions for RESTful OPC UA clients in IIoT environments," Siemens AG, University of Stuttgart.

[43] OPC Foundation, "OPC UA Specification – Part 1: Overview and Concepts." [Online]. Verfügbar unter: https://reference.opcfoundation.org/Core/Part1/v105/docs/6. Abgerufen am: [04.02.25].

[44] Ascolab, "Basisdienste der OPC Unified Architecture." [Online]. Verfügbar unter: https://www.ascolab.com/de/unified-architecture/basisdienste.html. Abgerufen am: [04.02.25].

[45] OPC Foundation, "OPC UA Specification – Part 4: Services, Section 5.13.1." [Online]. Verfügbar unter: https://reference.opcfoundation.org/Core/Part4/v104/docs/5.13.1. Abgerufen am: [04.02.25].

[46] W. Sim, B. Song, J. Shin und T. Kim, "Data Distribution Service Converter based on the Open Platform Communications Unified Architecture Publish–Subscribe Protocol," Electronics, vol. 10, no. 20, p. 2524, Oct. 2021. DOI: 10.3390/electronics10202524.

[47] GraphQL Foundation, "Queries." [Online]. Verfügbar unter: https://graphql.org/learn/queries/. Abgerufen am: [03.02.25].

[48] Apollo GraphQL, "Resolvers." [Online]. Verfügbar unter: https://www.apollographql.com/docs/apollo-server/data/resolvers. Abgerufen am: [03.02.25].

[49] Moldstud, "Implementing GraphQL Subscriptions for Real-Time Collaboration." [Online]. Verfügbar unter: https://moldstud.com/articles/p-implementing-graphql-subscriptions-for-real-time-collaboration. Abgerufen am: [13.02.25].

[50] Apollo GraphQL, "Subscriptions." [Online]. Verfügbar unter: https://www.apollographql.com/docs/apollo-server/data/subscriptions. Abgerufen am: [13.02.25].

[51] InterviewPro, "App Module (app.module.ts) in Angular." [Online]. Verfügbar unter: https://medium.com/@interviewpro/app-module-ts-in-angular-c8e2068419ce. Abgerufen am: [13.02.25].
[52] OPC Foundation, "OPC UA Specification Part 6: Mappings - Section 7." [Online]. Verfügbar unter: https://reference.opcfoundation.org/Core/Part6/v104/docs/7. Abgerufen am: [15.02.25].
[53] O. Goodnews, "A Model for Optimizing the Runtime of GraphQL Queries," ResearchGate, 2022. [Online]. Verfügbar unter: https://www.researchgate.net/profile/Ogboada-Goodnews/publication/358692438_A_Model_for_Optimizing_the_Runtime_of_GraphQL_Queries/links/620f36ddf02286737ca83570/A-Model-for-Optimizing-the-Runtime-of-GraphQL-Queries.pdf. Abgerufen am: [16.02.25].
[54] GraphQL Foundation, "Introspection." [Online]. Verfügbar unter: https://graphql.org/learn/introspection/. Abgerufen am: [16.02.25].
[55] AWS (2023). "Was ist Caching?" Verfügbar unter: https://aws.amazon.com/de/caching/
[56] RestfulAPI (2023). "Caching in RESTful APIs." Verfügbar unter: https://restfulapi.net/caching/
[57] - Divakaran, J. (2021). "Optimizing API Performance: A Comparative Study of REST and GraphQL." Verfügbar unter: https://www.diva-portal.org/smash/get/diva2:1571154/FULLTEXT01.pdf
[58] - GraphQL Foundation (2023). "Caching in GraphQL." Verfügbar unter: https://graphql.org/learn/caching/
[59] Apollo GraphQL (2023). "Cache Configuration in Apollo Client." Verfügbar unter: https://www.apollographql.com/docs/react/caching/cache-configuration
[60] Microsoft, "API design best practices." [Online]. Verfügbar unter: https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design.
[61] IBM, "Seven key insights on GraphQL trends." [Online]. Verfügbar unter: https://www.ibm.com/think/insights/seven-key-insights-on-graphql-trends. Abgerufen am: [03.02.25].
[62] Apollo GraphQL, "Caching in Apollo Server." [Online]. Verfügbar unter: https://www.apollographql.com/docs/apollo-server/performance/caching. Abgerufen am: [15.02.25].