Hochschule Merseburg

Fachbereich Ingenieur- und Naturwissenschaften

Studiengang Angewandte Informatik

Bachelorarbeit zum Thema:

Entwicklung einer Web-API zum Zugriff auf OPC UA Server

zur Erlangung des akademischen Grades

Bachelor of Science

vorgelegt von: Marcel Schalk

Matrikelnummer: 23828

Erstgutachter: Prof. Dr.-Ing. Thomas Meier

Zweitgutachter: Prof. Dr. Peter Helm

Abgabedatum: 04.11.2021

Inhaltsverzeichnis

Inhaltsverzeichnis ..... **II**

Abbildungsverzeichnis ..... **V**

Tabellenverzeichnis ..... **VI**

Listings ..... **VI**

Abkürzungsverzeichnis ..... **VII**

Glossar ..... **VII**

1 Einleitung **1**

1.1 Gegenstand und Motivation 1

1.2 Anforderung und Zielsetzung 1

1.3 Gliederung der Arbeit 2

1.4 Abgrenzung der Arbeit 2

2 Grundlagen **3**

2.1 OPC UA 3

2.1.1 OPC Foundation 3

2.1.2 Gründe für die Entwicklung von OPC UA 3

2.1.3 Grundlagen von OPC UA 4

2.1.3.1 Aufbau OPC UA 5

2.1.3.2 Address Space Model 6

2.1.3.3 Datentypen 9

2.1.3.4 Namespaces und NodeIds 11

2.1.3.5 Views 12

2.1.3.6 Services 13

2.2 Web-API 15

2.3 HTTP 16

2.3.1 Request-Response-Prinzip 16

2.3.1.1 URI 16

2.3.1.2 Request 17

2.3.1.3 Response 17

2.3.2 HTTP-Methoden 18

2.3.3 HTTP-Statuscodes 20

3 Systematische Analyse **22**

3.1 Istzustand 22

3.2 Sollzustand 22

3.3 Anwendungsfallanalyse 23

3.4 Anforderungsanalyse ... 24
3.4.1 Funktionale Anforderungen ... 25
3.4.2 Nicht-funktionale Anforderungen ... 29
4 Konzeption ... 31
4.1 Auswahl der Hilfsmittel ... 31
4.1.1 Programmiersprache ... 31
4.1.2 SDK ... 32
4.1.3 Webframework ... 32
4.1.4 Entwicklungsumgebung ... 32
4.1.5 API-Dokumentation ... 33
4.2 Auswahl der Request/Response-Technologie ... 33
4.2.1 Vergleich ... 34
4.2.1.1 REST ... 34
4.2.1.2 JSON:API ... 38
4.2.1.3 gRPC ... 41
4.2.1.4 GraphQL ... 43
4.2.1.5 Verbreitung und Nutzung ... 46
4.2.2 Festlegung ... 49
4.3 Auswahl Push-Technologien ... 50
4.3.1 Vergleich ... 50
4.3.1.1 HTTP Polling ... 51
4.3.1.2 HTTP Long Polling ... 52
4.3.1.3 WebSockets ... 53
4.3.1.4 Server-Sent Events (SSE) ... 54
4.3.1.5 Push API ... 55
4.3.1.6 Verbreitung und Nutzung ... 56
4.3.2 Entscheidung ... 57
4.4 Auswahl des Authentifizierungsverfahrens ... 57
4.5 Repräsentationsformat ... 58
4.6 Ressourcen- & URI-Design ... 59
4.7 Abbildung der funktionalen Anforderungen ... 61
4.7.1 Verbindungsaufbau zum Server ... 61
4.7.2 Anzeige und Navigation ... 62
4.7.3 Attribute lesen/schreiben ... 62
4.7.4 Historische Werte lesen ... 62
4.7.5 Query ... 63
4.7.6 Funktionen aufrufen ... 63
4.7.7 Subscriptions ... 63

5 Implementierung ... 64
5.1 Projektstruktur ... 64
5.2 Logging und Debugging ... 64
5.3 Nutzung von Express ... 65
5.3.1 app.js ... 65
5.3.2 router.js ... 65
5.3.3 express-ws ... 66
5.3.4 Basic Auth ... 67
5.4 Controller ... 67
5.4.1 AttributeDetail.js ... 68
5.4.2 getPossibleAttributes.js ... 69
5.4.3 getNode.js ... 69
5.4.4 callMethod.js & getMethod.js ... 69
5.4.5 getAttribute.js ... 70
5.4.6 writeAttribute.js ... 70
5.4.7 getHistory.js ... 71
5.4.8 getReferences.js & getMethods.js ... 71
5.4.9 getReference.js ... 71
5.4.10 Subscription.js ... 72
5.4.11 Queries ... 72
6 Ergebnisse und Ausblick ... 73
6.1 Beschreibung und Bewertung der Vorgehensweise ... 73
6.2 Zusammenfassung ... 73
6.3 Ausblick & Fazit ... 73
Anhang ... 75
A Übersicht der UA-Node-Klassen ... 75
B Hierarchische Darstellung der eingebauten Datentypen ... 75
C Exemplarische Abbildung der Ressourcen in JSON-HAL ... 76
C.1 Node ... 76
C.2 Attribut ... 77
C.3 Referenzen ... 77
C.4 Referenz ... 78
C.5 Methoden ... 78
C.6 Methode ... 79
D Begleit-DVD ... 80
7 Literaturverzeichnis ... 81

Abbildungsverzeichnis

Abbildung 1: Aufbau OPC UA 5
Abbildung 2: OPC UA Object Model 6
Abbildung 3: Schematische Darstellung des Node Model 7
Abbildung 4: Einsatzmöglichkeiten von Views an einem Beispiel 12
Abbildung 5: URI-Syntax 16
Abbildung 6: Use-Case-Diagramm für die Web-API 24
Abbildung 7: Nutzung von HTTP/2 nach Servern 42
Abbildung 8: Interesse an den Web-APIs nach Google Trends 46
Abbildung 9: Bevorzugte API-Designs laut SmartBear 47
Abbildung 10: Nutzung von API-Designs laut RapidAPI 48
Abbildung 11: Vertrautheit mit API-Technologien 49
Abbildung 12: Ablauf von HTTP Polling 51
Abbildung 13: Ablauf von HTTP Long Polling 52
Abbildung 14: Beispielhafte Kommunikation WebSockets 53
Abbildung 15: Ablauf Server-Sent Events 54
Abbildung 16: Schematische Darstellung der Kommunikation bei der Push API 55
Abbildung 17: Google Trends Chart zu Push-Technologien 56

Tabellenverzeichnis

Tabelle 1: Zusammenstellung der Attribute die alle Nodeklassen gemein haben 8

Tabelle 2: Aufbau einer NodeId 11

Tabelle 3: Eigenschaften der HTTP-Methoden 20

Tabelle 4: Abbildung der HTTP-Verben auf die CRUD-Methoden 35

Tabelle 5: Auflistung der Ressourcen 60

Listings

Listing 1: Beispiel strukturierte Daten in Pseudocode 11

Listing 2: Beispielhafte HTTP-Anfrage 17

Listing 3: Beispielhafte HTTP-Antwort 17

Listing 4: Beispiel-Code für HATEOAS 35

Listing 5: Code-Beispiel Request/Response mit REST 36

Listing 6: Code-Beispiel Request/Response mit JSON:API 39

Listing 7: Code-Beispiel Sparse Fields JSON:API 40

Listing 8: Beispielhafte Protobuf-Datei 41

Listing 9: Beispiel für ein Schema unter GraphQL 43

Listing 10: Beispiel für eine Request/Response bei GraphQL 44

Listing 11: Beispielhafte Darstellung des JSON-HAL-Objekts Item1 58

Listing 12: Gekürzte Darstellung der Datei app.js 65

Listing 13: Beispielhafte Darstellung einer Anfrage in Express 65

Listing 14: Beispielhafte Darstellung der Einbindung von WebSockets 66

Listing 15: Beispiel-Controller 67

Abkürzungsverzeichnis

ERP	Enterprise Resource Planning
GUID	Globally Unique Identifier
JSON	Java Script Object Notation
MES	Manufacturing Execution System
SDK	Software Development Kit
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
XML	Extensible Markup Language

Glossar

Boolean	Element das die Zustände *wahr* oder *falsch* annehmen kann
Double	Eine Gleitkommazahl mit der doppelten Anzahl an Bytes
Framework	Programmiergerüst für Entwickler zur Programmierung
Hash	Ein Algorithmus der Daten auf einen Wert fester Länge abbildet
Int32	Datentyp einer Ganzzahl mit dem Bereich -2147483648 bis 2147483647
Opaque	Ein binär kodierter Wert
Paginierung	Darstellung einer Menge an Daten verteilt über mehrere Seiten
String	Endliche Folge von Zeichen aus einem festgelegten Zeichensatz
Subscription	Im Kontext dieser Arbeit ein Abonnement von Daten
UInt32	Ganzzahl ohne Vorzeichen mit einem Bereich von 0 bis 4294967295

1 Einleitung

1.1 Gegenstand und Motivation

OPC UA ist der Nachfolger von OPC und stellt einen plattformunabhängigen, serviceorientierten Standard zum Austausch von Daten dar. Der 2007 verabschiedete Standard genießt aufgrund der Kompatibilität zu OPC und der zukunftsfähigen Architektur eine hohe Akzeptanz in der Industrie 4.0. [1]

OPC UA unterstützt die Kommunikation über TCP sowie eine verschlüsselte Kommunikation über HTTPS mit Hilfe von TLS. Bis ins Jahr 2015 war es seitens des Standards angedacht, SOAP über HTTP zu nutzen. Da SOAP jedoch nur wenig Anklang in der Industrie fand, wurde diese Möglichkeit mit Version 1.03 als *deprecated* (veraltet) markiert. [2, S. 45]

Erst zum Ende des Jahres 2017 wurden mit der Version 1.04 neue Möglichkeiten der Netzwerkkommunikation eingeführt. Moderne Technologien und Standards wie JSON, WebSocket und eine Option Services ohne Session zu nutzen, hielten Einzug. [2, S. 60–61]

Damit wurde es einer Vielzahl von Webanwendungen möglich, mit OPC UA zu arbeiten. Das Problem an diesen neuen Lösungen ist, dass sie noch keine große Anwendung finden, da entweder noch nicht alle Systeme auf die neuste Version aktualisiert wurden oder noch keine Implementierung der Features stattgefunden hat. Außerdem bedürfen viele Webanwendungen zusätzlich noch einer entsprechenden Schnittstelle, um mit einem OPC UA Server kommunizieren zu können.

1.2 Anforderung und Zielsetzung

Ziel ist es eine API zu entwickeln, die dazu in der Lage ist die neuen Features auch auf ältere Implementierungen von OPC UA abzubilden, die die neuen Technologien noch nicht unterstützen bzw. implementiert haben. Ein vorgegebener Einsatzbereich bei dem die API Anwendung finden soll, ist ein OPC UA Server in den Laboren der Hochschule Merseburg, auf den ein Zugriff durch die API ermöglicht werden soll.

Weiterhin muss diese API auf moderne Webtechnologien setzen und es soll ein Vergleich dieser Ansätze stattfinden. Technologien wären hierbei beispielsweise WebSockets, REST oder GraphQL.

1.3 Gliederung der Arbeit

Im ersten Teil der Arbeit werden Grundlagen zu den Themen OPC UA, Web-API und HTTP vermittelt, die für das Verständnis der Arbeit entscheidend sind. Anschließend findet eine systematische Analyse statt, bei der die Ziele dieser Arbeit genauer beleuchtet werden und aus denen Anforderungen an das Projekt formuliert werden. In der darauffolgenden Konzeption wird ein Vergleich der möglichen Technologien zur Umsetzung angestellt und eine Auswahl getroffen. Anschließend wird in der Implementierung eine Umsetzung anhand der vorher getroffenen Entscheidungen aufgezeigt. Der letzte Teil der Arbeit umfasst die Auswertung in Form der Präsentation der Ergebnisse und eines Ausblicks.

1.4 Abgrenzung der Arbeit

In dieser Arbeit kann nicht auf alle für diese Umsetzung geeigneten Technologien eingegangen werden, weshalb die Auswahl auf die in dieser Arbeit genannten Technologien begrenzt wurde. Diese Auswahl stützt sich zum einen auf vorhandene Literatur und auf angeführte Statistiken zur Verbreitung der Technologien.

In dieser Arbeit wird kein vollständig implementiertes Projekt vorgelegt. Es werden vielmehr Ansätze der Implementierung aufgezeigt. Außerdem wird diese Arbeit keine Grundlagen zur verwendeten Programmiersprache vermitteln.

2 Grundlagen

2.1 OPC UA

OPC UA steht für Open Platform Communications Unified Architecture und ist ein Standard für den plattformunabhängigen Austausch von Daten [3].

Im Folgenden wird auf die Entstehung von OPC UA eingegangen und es wird einen Überblick in die Technik gegeben.

2.1.1 OPC Foundation

Die OPC Foundation wurde im Jahr 1994 als Nonprofit-Organisation mit dem Ziel gegründet einen *PnP*-Standard (Plug-and-play) für Gerätetreiber zu entwickeln, die einen standardisierten Zugriff auf Daten von Automatisierungssystemen ermöglichen [4]. Als Resultat dieser Bestrebungen wurde 1996 OPC Data Access verabschiedet. Beinahe alle Anbieter von industrieller Automatisierung wurden Mitglieder der OPC Foundation, was unter anderem daran lag, dass die OPC Foundation praxisrelevante Standards schneller implementieren konnte als andere Organisationen. Dies war möglich, da man sich bei OPC auf wichtige Funktionen fokussiert hatte und verbreitete Technologien wie *COM* und *DCOM* (Techniken zur Interprozesskommunikation) als Basis für die Kommunikation nutzte. [5, S. 1]

Das sorgte dafür, dass die OPC Foundation nach nur 12 Jahren 450 Mitglieder hatte und über 15000 Geräte auf dem Markt waren, die den Standard nutzten [5, S. 2].

2.1.2 Gründe für die Entwicklung von OPC UA

OPC war ein großer Erfolg, was die Verbreitung in der Industrie zeigt. Es wird auf verschiedensten Ebenen der Automatisierungstechnik eingesetzt sowie in Bereichen, für die OPC ursprünglich nicht konzipiert wurde. Mit der Zeit eröffneten sich immer mehr Möglichkeiten. Es gab jedoch auch Bereiche, in denen OPC nicht eingesetzt werden konnte, obwohl der Bedarf seitens der Hersteller und Entwickler bestand. Der Hauptgrund dafür war der Einsatz von COM und DCOM. [5, S. 8]

Durch die Nutzung von COM und DCOM waren die Entwickler und Hersteller Einschränkungen unterlegen. Zwar wurde durch diese Technologien die komplette Infrastruktur für die Kommunikation inklusive der Sicherheitsdienste wie Authentisierung, Autorisierung und Verschlüsselung zur Verfügung gestellt, gleichzeitig waren Sie aber auch kompliziert in der Verwendung. Probleme mit der Implementierung über COM und DCOM stellten sogar den Spitzenreiter in den Statistiken zu Supportanfragen bei den Herstellern dar. Ein weiteres

Problem ergab sich durch lange und nicht konfigurierbare Zeiten bei der Erkennung von Kommunikationsabbrüchen. Dies kann dazu führen, dass mehrere Sekunden vergehen, bis der Client über einen Verbindungsabbruch informiert wird, was für die meisten industriellen Anwendungen unzureichend ist. [6, S. 97]

Neben diesen Problemen gab es auch Hürden bei der Firewall-übergreifenden OPC-Kommunikation. Da DCOM die genutzten Ports zufällig festlegt, muss eine Vielzahl von Ports in der Firewall geöffnet werden, womit sich mehr Sicherheitslücken für Angreifer ergeben. Unter *NAT* (Netzwerkadressübersetzung) basierten Firewalls ließ sich sogar überhaupt keine Verbindung herstellen, da DCOM nicht mit der Adressumsetzung umgehen kann. Eine Möglichkeit um dieses Problem zu umgehen war OPC-Tunneling, was aber eine weitere Komponente darstellt, die auf der Hardware ausgeführt werden muss.[6, S. 98]

Das größte Problem, dass durch COM und DCOM entstand, war die Bindung an Windows, weswegen die plattformübergreifende Arbeit erschwert wurde. OPC XML-DA war der erste Versuch den OPC Standard zu erweitern. Anstelle von COM und DCOM wurden Techniken wie *SOAP* über HTTP und Webservices eingesetzt, wodurch eine plattformunabhängige Kommunikation ermöglicht wurde, während die Funktionalität von OPC Data Access erhalten blieb. Der erwartete Erfolg blieb jedoch aus, was vor allem an dem fünf- bis siebenfach geringeren Datendurchsatz im Vergleich zu DCOM sowie die Begrenzung auf lediglich acht Services von OPC Data Access lag. [5, S. 7–8], [6, S. 99]

2.1.3 Grundlagen von OPC UA

Die Zielstellung für OPC UA war es, einen Ersatz für OPC und dessen COM-basierten Spezifikationen zu schaffen, welcher gleichzeitig keine Funktionalität oder Performance einbüßt. Zusätzlich sollte der Standard plattformunabhängig sein und ein vielseitiges und erweiterbares Datenmodell anbieten, um komplexe Systeme abzubilden. Um diese Ziele zu erfüllen, wurde im Jahr 2006 die erste Version von OPC UA spezifiziert. [7]

Im weiteren Verlauf dieses Kapitels wird grob der Aufbau und die Funktionsweise von OPC UA aufgezeigt.

2.1.3.1 Aufbau OPC UA

OPC UA besteht aus mehreren Ebenen bzw. Teilen die in Abbildung 1 dargestellt sind.

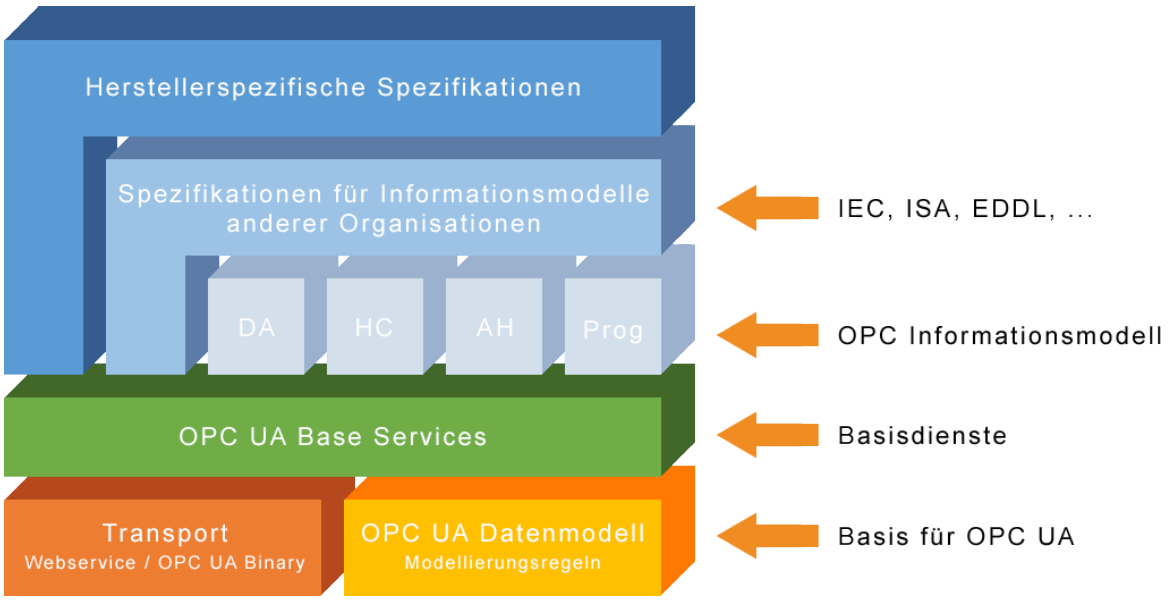


Abbildung 1: Aufbau OPC UA (Quelle: eigene Darstellung in Anlehnung an [3])

Transport

Beschreibt die Kommunikation über verschiedene Technologien wie TCP, Webservices, XML und HTTP. [5, S. 10–11]

Datenmodell

Das Datenmodell ist das Fundament von OPC UA. Es legt die Regeln fest, nach denen OPC UA aufgebaut ist. Es definiert die Zugangspunkte zum Adressraum, über die alle Daten abgerufen werden können, und legt die Basistypen fest, um eine Typhierarchie aufzubauen. [5, S. 10–11]

Services

Die Services bilden die Schnittstellen zwischen einen Server und einem Client. Der Server ist hierbei der Anbieter eines Informationsmodells und der Client der Konsument des Informationsmodells. [5, S. 10–11]

Informationsmodell

Um die bewährte Funktionalität des klassischen OPC zu erhalten, wurden OPC Data Access (DA), Alarm & Conditions (AC), Historical Access (HA) und Programs (Prog) übernommen. Das Informationsmodell wird erweitert durch Informationsmodelle anderer Organisationen, wie zum Beispiel PLCOpen, FDI, PROFInet und viele mehr. Um OPC UA dynamischer zu gestalten ist es möglich, das Informationsmodell durch herstellerspezifische Spezifikationen zu erweitern. [5, S. 10–11]

2.1.3.2 Address Space Model

In diesem Kapitel wird auf die Konzepte des OPC UA Address Space, was auch oft als das *Meta Model* (übergeordnetes Modell zur Formalisierung anderer Modellkonzepte) von OPC UA bezeichnet wird, eingegangen [5, S. 81].

2.1.3.2.1 Object Model

Das Objektmodell beschreibt den Aufbau von Objekten in Bezug auf Variablen und Methoden und ermöglicht die Abbildung von Beziehungen zu anderen Objekten. Zusätzlich können Objekte typisiert werden. Einen vergleichbaren Ansatz findet man bei objektorientierten Programmiersprachen, wie zum Beispiel Java mit seinen Klassen und den dazugehörigen Member-Variablen und -Methoden. [8, S. 5]

In Abbildung 2 wird aufgezeigt, wie man mit verschiedenen Services auf Objekte zugreifen kann bzw. wie man mit Ihnen arbeiten kann.

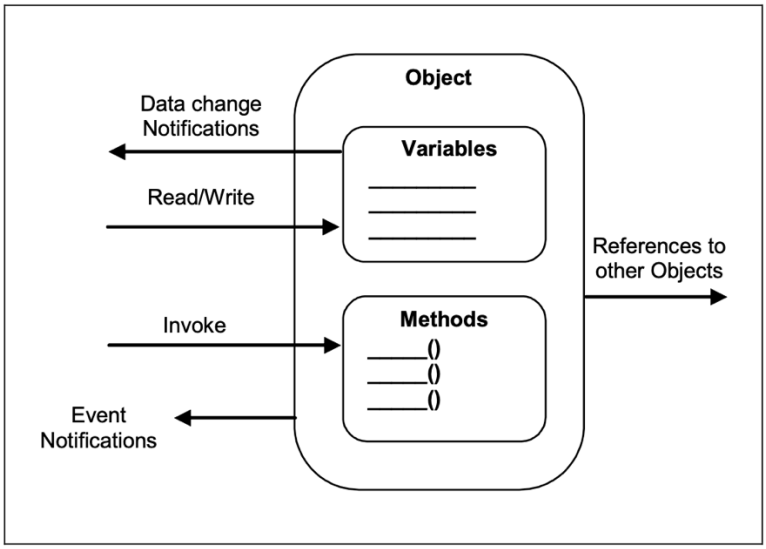


Abbildung 2: OPC UA Object Model (Quelle: [8, S. 5])

2.1.3.2.2 Node Model

Jedes Objekt sowie deren Komponenten, werden im Adressraum als sogenannte *Nodes* (Knoten) repräsentiert. Diese Nodes sind Instanzen der acht Node-Klassen, die durch die OPC Foundation festgelegt wurden. Eine Übersicht der Node-Klassen befindet sich im Anhang A. Clients und Servern ist es nicht erlaubt weitere Node-Klassen zu definieren [8, S. 24].

In Abbildung 3 wird das Node Model schematisch dargestellt.

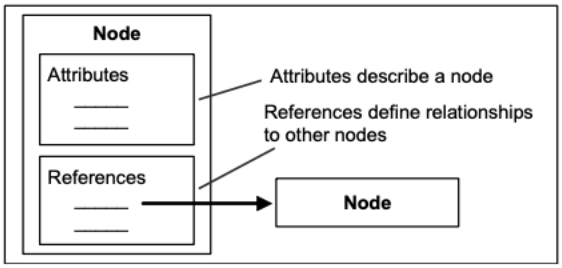


Abbildung 3: Schematische Darstellung des Node Model (Quelle: [8, S. 6])

Die Node-Klassen definieren sich durch die Attribute und Referenzen, sodass die Nodes einer Klasse die gleichen Attribute und Referenzen aufweisen. Attribute sind in dem Fall Eigenschaften, welche die Node beschreiben. In Tabelle 1 befindet sich eine Zusammenstellung der Attribute, die alle Node-Klassen gemein haben. [8, S. 6]

Eine komplette Liste aller Attribute von OPC UA befindet sich in *Annex A.1 Attribute Ids* der OPC-UA-Spezifikation Teil 6[1].

[1] [2, S. 61]

Tabelle 1: Zusammenstellung der Attribute die alle Nodeklassen gemein haben (Quelle: in Anlehung an [5, S. 23])

Attribut	Beschreibung	Beispiel
NodeId	Ein einzigartiger *Identifier* (Bezeichnung) innerhalb eines OPC UA Servers, der genutzt wird, um die Nodes mittels eines Service zu adressieren.	ns=6;s=MyDevice
NodeClass	Die zugehörige Node-Klasse.	Object
BrowseName	Kennzeichnet die Node beim Durchsuchen des OPC UA Servers mit der browse-Funktion.	6:MyDevice
DisplayName	Ein lokalisierter Anzeigename, der genutzt werden sollte in Benutzeroberflächen.	Locale: en Text: "MyDevice"
Description	Ein lokalisiertes Attribut zum Beschreiben der Node.	Locale: en Text: "It's my device"
WriteMask	Dieses Attribut gibt an, welche Attribute der Node allgemein modifizierbar sind.	None
UserWriteMask	Dieses Attribut gibt an, welche Attribute der Node modifizierbar durch den aktuell verbundenen Nutzer.	None

Referenzen dienen zum Verknüpfen der Nodes. Im Gegensatz zu Attributen handelt es sich bei Referenzen um Instanzen von Referenzen der Node-Klasse ReferenceType. Die Quell-Node, welche die Referenz enthalt, die Ziel-Node und der Referenztyp werden genutzt, um die Referenz eindeutig zu bestimmen. Das heißt, dass eine Verbindung zwischen zwei Nodes nicht mehrfach mit demselben Referenztyp vorkommen kann. [8, S. 6–7]

2.1.3.2.3 Variablen

Variablen werden dazu genutzt Werte zu repräsentieren. OPC UA bietet dazu zwei Typen von Variablen an. Die zwei nachfolgend genannten Typen unterscheiden sich in der Art der Daten, die sie darstellen und danach, ob sie andere Variablen beinhalten können. [8, S. 7]

Properties

Properties sind vom Server definierte Eigenschaften von Objekten und anderer Nodes. Sie werden genutzt, um darzustellen, was die Node repräsentieren soll, wie zum Beispiel eine Auftragsnummer. Im Gegensatz zu Attributen, welche durch die Spezifikationen vorgegeben sind und von allen Nodes einer Node-Klasse geteilt werden,

können Properties durch den Server definiert werden. Während ein Attribut beispielsweise den Datentyp einer Variablen deklariert, kann ein Property dazu genutzt werden die Maßeinheit einer Variablen zu definieren. [8, S. 7]

Datenvariablen

Datenvariablen stehen für den Inhalt eines Objekts. Beispiele hierfür sind die Temperatur eines Temperatursensors oder der Durchfluss einer Durchflussmessung. Variablen eignen sich besonders für komplexe Daten, da sie Teilvariablen haben können. [5, S. 58]

Beispielsweise kann die Konzentration eines bestimmten Stoffes in einem Produkt über den Leitwert berechnet werden. Während die Variable für die Konzentration für die Anzeige im Prozessleitsystem genutzt wird, beinhaltet sie die Teilvariable für den Leitwert zur Berechnung der Konzentration.

Bei der Erstellung kann sich die Entscheidung zwischen Property und Datenvariable als ein schwieriges Thema erweisen [5, S. 58]. Bei der Modellierung sollte man sich danach richten, wie oft der Wert der Variable geändert wird. Properties sind eher statische Werte, die selten oder nie geändert werden, während Datenvariablen oft geändert werden können, jedoch nicht geändert werden müssen. Dies ist nicht in den Spezifikationen festgehalten und gibt lediglich ein Leitpfaden vor, da nicht genau definiert werden kann was als „oft“ oder „selten“ angesehen wird. Eine weitere Möglichkeit ist die Betrachtung, ob es sich bei der Variable um Konfigurationsdaten handelt. Ist dies der Fall, kommt eher ein Property in Frage. [5, S. 59]

Datenvariablen können komplexere Daten abbilden, sind dadurch aber auch schwerer zu handhaben. Sollte keine der vorher genannten Eigenschaften bei der Einordnung behilflich sein, kann man sich daher auch über den möglichen Einsatz Gedanken machen. Werden die erweiterten Möglichkeiten einer Datenvariablen nicht benötigt, kann sich der Einsatz von Properties auf Grund der einfacheren Handhabung als sinnvoller erweisen. [5, S. 59]

2.1.3.3 Datentypen

In OPC UA werden Daten als Werte von Variablen, Argumente von Methoden oder Event-Felder ausgetauscht. Diese Daten können von einer Vielzahl verschiedener Datentypen sein. Zusätzlich zu diesen Datentypen bietet OPC UA die Attribute *ValueRank* und *ArrayDimension* an, welche definieren, ob der Datentyp in skalarer Form, in Form eines einfachen Arrays oder als multidimensionales Array vorliegt, wobei skalar bedeutet, dass die Variable einen einzelnen Wert darstellt. [5, S. 61]

Datentypen werden im Adressraum als Nodes der Node-Klasse *DataType* dargestellt. Diese Nodes bilden eine Typhierarchie, ausgehend von der Node *BaseDataType*. Dies ermöglicht es dem Server neue Datentypen zu definieren, die durch Referenzen in die Hierarchie der Datentypen eingebunden sind. [5, S. 61]

Die Datentypen von OPC UA lassen dabei sich in folgende vier Gruppen einordnen:

Eingebaute Datentypen

Diese Datentypen sind durch die OPC-UA-Spezifikationen vorgegeben und können im Gegensatz zu den anderen Datentypen nicht verändert oder erweitert werden. Zu diesen 25 Datentypen gehören Standarttypen wie Int32, Boolean oder Double, aber auch OPC-UA-spezifische Typen wie *NodeId* und *QualifiedName*. Eine hierarchische Übersicht dieser Datentypen befindet sich im Anhang B. [5, S. 63]

Simple Datentypen

Diese Datentypen sind Subtypen der eingebauten Datentypen, weshalb sie ohne zusätzliche Informationen gelesen werden können. Weitere Informationen zur Verarbeitung befinden sich im DataType-Attribut der Variable. Ein Beispiel hierfür ist die Dauer also der Abstand zwischen zwei Zeitpunkten, welche ein Subtyp von Double ist und einen Zeitintervall in Millisekunden angibt. [5, S. 62–63]

Enumerationen

Hierbei handelt es sich um eine Aufzählung von möglichen Zuständen, die eine Variable annehmen kann [5, S. 63]. Ein Anwendungsbeispiel ist eine Pumpe in der Industrie, welche die Zustände *In Betrieb*, *Bereit* und *Fehler* annehmen kann. Die Daten werden dabei als Int32 übertragen, welche den einzelnen Zuständen zugeordnet werden können. Die Zustände werden im Property *EnumString* als Array von lokalisierten Texten im Datentyp hinterlegt. [5, S. 63]

Strukturierte Datentypen

Diese Datentypen sind komplexe Datentypen, welche die mächtigsten Datentypen in OPC UA darstellen. Dies macht sie aber zugleich auch zu den kompliziertesten Datentypen, da der Server zu jedem dieser Datentypen Informationen bereitstellen muss, wie diese zu lesen sind. [5, S. 64]
Listing 1 zeigt ein Beispiel für den strukturierten Datentyp Argument, welcher für die Argumente einer Methode genutzt wird:

```
struct Argument {
    String      name;
    NodeId      dataType;
    Int32     valueRank;
    UInt32[]      arrayDimensions;
    LocalizedText     description;
};
```

Listing 1: Beispiel strukturierte Daten in Pseudocode (Quelle: Eigene Darstellung in Anlehnung an [9])

2.1.3.4 Namespaces und NodeIds

Namespaces, zu deutsch Namensräume, werden in OPC UA benutzt, um über verschiedene Server hinweg eindeutige Identifier zu erzeugen. *NodeIds* sind dieser eindeutige Identifier, den jede Node besitzt. Die NodeId setzt sich wie in Tabelle 2 zusammen. [5, S. 68]

Tabelle 2: Aufbau einer NodeId (Quelle: Eigene Darstellung in Anlehung an [5, S. 68])

Attribut	Datentypen
Namespace Index	UInt16
IdentifierType	Enumeration
Identifier	Numerisch, String, GUID, Opaque

Der *NamespaceIndex* ist eine Zahl, welche die URI eines Namespace-Servers repräsentiert. Die Namespace-Tabelle, in der jedem auf dem Server hinterlegten NamespaceIndex eine URI zugewiesen wird, ist über den Adressraum auslesbar. [10]

Die NodeId wird in der XML-Notation kodiert. Eine String-Notation wird genutzt, um die NodeId in kompakter Form auszugeben. Beispiele dafür sind:

- *ns=6;s=MyDevice*
- *ns=1;g= c4f5e12e-03e8-4a7a-919f-6aaf3d2e5005*
- *i=23*

Die Buchstaben *ns* stehen hierbei für Namespace. Der Buchstabe, nach dem Semikolon, gibt den Identifier-Typ an, wobei *s* für String, *i* für numerisch, *g* für GUID und *b* für Opaque steht. Im dritten Beispiel handelt es sich um den NamespaceIndex *0*, der für die URI *http://opcfoundation.org/UA/* reserviert ist. Auf Grund dieser Konvention kann dieser Index weggelassen werden. [11, S. 109].

2.1.3.5 Views

Views (Ansichten) sind eine Möglichkeit, eine eingeschränkte Menge an Nodes und Referenzen eines Adressraums auszugeben. Dies bietet sich bei großen Adressräumen an, bei denen Views für individuelle Einsatzzwecke bereitgestellt werden können. [5, S. 71]

Es gibt zwei Möglichkeiten diese Views zu nutzen. Die erste Möglichkeit besteht darin, die View als eine Art Filter ausgehend von einer Start-Node zu nutzen. Bei der zweiten Möglichkeit organisiert die View alle zugehörigen Elemente selbst und dient dabei als Zugangspunkt. Auch eine Kombination aus beiden Optionen ist möglich. [5, S. 72–73]

In Abbildung 4 ist dies an einem Beispiel verdeutlicht. Die Views Synthese und Extraktion, die wie alle Instanzen über die Start-Node *Objects* referenziert werden, sind die Zugangspunkte zu den einzelnen Regelgruppen und ihren dazugehörigen Elementen. Die View Ventile hingegen referenziert die Ventile und wirkt als Filter für Elemente, die bereits unter einer anderen View organisiert werden.

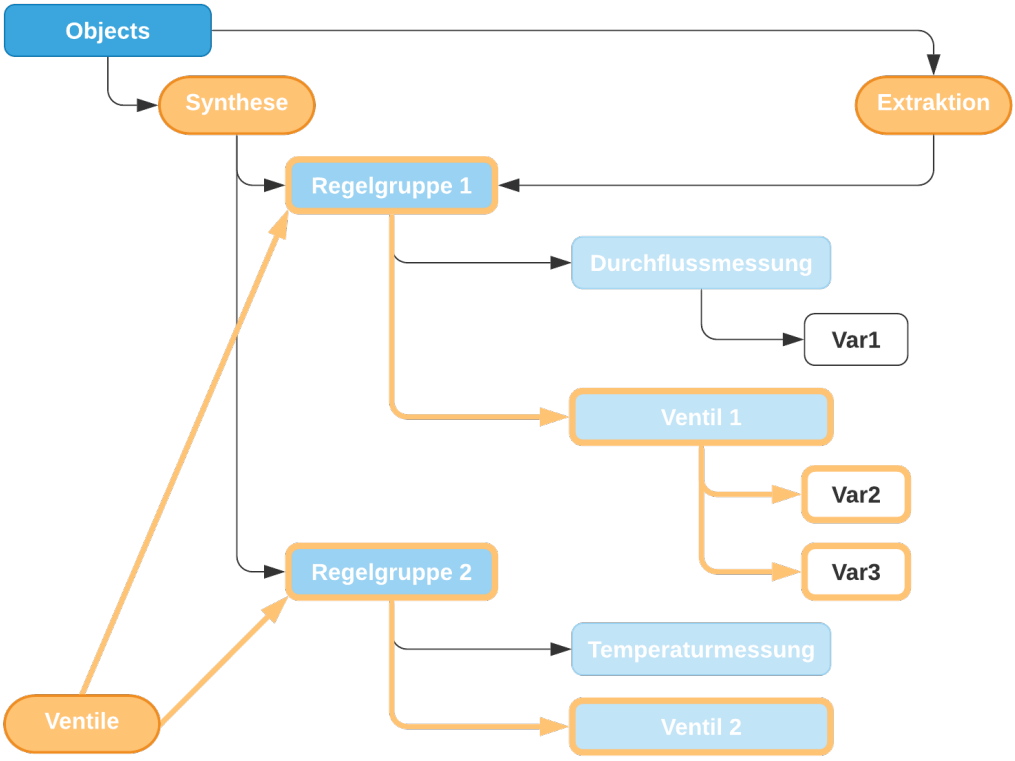


Abbildung 4: Einsatzmöglichkeiten von Views an einem Beispiel (Quelle: eigene Darstellung)

2.1.3.6 Services

OPC UA bietet eine Reihe von Services, um mit dem Informationsmodell zu arbeiten. Diese Services arbeiten nach dem *Request-Response-Prinzip* (Anfrage-Antwort-Prinzip) und sind unabhängig vom eingesetzten Transportprotokoll [5, S. 125], [12, S. 7–8].

Diese Services werden in zehn *Service Sets* (Gruppen) eingeordnet. Im folgenden Kapitel wird kurz auf die Service Sets und deren wichtigste Services eingegangen, um ein Grundverständnis zu vermitteln. Eine komplette Übersicht aller Services lässt sich in der OPC-UA-Spezifikation *OPC 10000-4: OPC Unified Architecture Part 4: Services*[2] finden.

Ein Server muss nicht alle dieser Service Sets unterstützen und kann auch nur Gebrauch von einzelnen Services innerhalb eines Sets machen. Eine Liste der unterstützten Services lässt sich dem Profil des Servers entnehmen. [12, S. 8]

Für einen tieferen Einblick in die Thematik Profile wird auf die OPC-UA-Spezifikationen *OPC 10000-7: OPC Unified Architecture Part 7: Profiles*[3] verwiesen.

Discovery Service Set

Dieses Service Set stellt Services bereit, um Server zu finden und deren Endpunkte zu bestimmen. Die zurückgegebenen Endpunkte stellen alle wichtigen Informationen bereit die notwendig sind, um eine sichere Verbindung und eine *Session* (Sitzung) zwischen dem Client und einem Server aufzubauen.[5, S. 133], [12, S. 9–13]

SecureChannel Service Set

Durch diese Services ist es möglich, eine sichere Verbindung aufzubauen bzw. zu schließen, durch welche die Vertraulichkeit und Integrität der Daten gewährleistet werden soll. [12, S. 18]

Session Service Set

Diese Services ermöglichen den Aufbau und das Schließen einer Session zwischen einem Client und einem Server. Mit dem Service *ActivateSession* wird außerdem die Möglichkeit geboten, den Nutzer der Session festzulegen und der Session einen neuen *SecureChannel* zuzuweisen. [12, S. 23–27]

---

[2] [12]

[3] [13]

NodeManagement Service Set

Das Service Set stellt Services zur Verfügung, um Nodes und Referenzen im Adressraum hinzuzufügen bzw. zu löschen. [12, S. 31]

View Service Set

Der Service *Browse* aus diesem Service Set dient zum Navigieren durch den Adressraum oder eine View. Hierbei wird dem Service eine Start-Node gegeben und das Ergebnis ist eine Liste aller referenzierten Nodes, wodurch es möglich ist, schrittweise eine Baumstruktur aufzubauen. Neben einem weiteren Service, der dazu dient, die NodeId anhand Browse-Pfades zu ermitteln, gibt es auch den Service *RegisterServer*. Damit ist es Clients möglich dem Server mitzuteilen, dass auf eine bestimmte NodeId oft zugegriffen wird. Der Server kann diesen Zugriff optimieren, indem er statt der eigentlichen NodeId kürzere numerische NodeIds zur Verfügung stellt, mit denen der Client arbeiten kann. Dies optimiert die Adressierung, da OPC UA numerische NodeIds effizienter verarbeiten kann. [5, S. 144], [12, S. 37–42]

Query Service Set

Der Browse-Service aus dem View Service Set ist vor allem für den Gebrauch in bekannten und kleinen Adressräumen gedacht. In großen Adressräumen mit umfangreichen Informationsmodellen oder Adressräumen, welche sehr dynamisch sind, wie man sie bei MES oder ERP-Systemen vorfinden, wird dafür das Query Service Set genutzt. Die Services nutzen Filter, um die abgefragten Nodes und Informationen einzugrenzen. Neben einer Start-Node oder View für die Abfrage und dem Typ der Objekte oder Variablen, kann auch angegeben werden, welche Daten zurückgegeben werden sollen. [5, S. 186–187]

Attribute Service Set

Diese Services dienen zum Lesen oder Verändern der Ist-Werte und Langzeitdaten von Attributen aus einer oder mehrerer Nodes. [12, S. 49–56]

Method Service Set

Das Service Set besteht nur aus einem Service mit der Bezeichnung *Call*, der zum Aufrufen von Methoden dient. [12, S. 58]

MonitoredItem Service Set

Dieses Service Set ist neben dem *Subscription Service Set*, dass im Folgenden behandelt wird, die Basis für Benachrichtigungen und die Überwachung von Elementen. Mit dem MonitoredItem Service Set wird festgelegt, welche Elemente überwacht werden, in welchem Intervall die Elemente abgetastet werden, sowie die Möglichkeit zum Filtern der Ergebnisse der Abfrage. In welchem Intervall diese Ergebnisse an einen Client gesendet werden, wird über das *Subscription Service Set* definiert. [12, S. 60–71]

Subscription Service Set

Wie bereits im vorangegangenen Absatz erwähnt, dient das Subscription Service Set zum Versenden von Daten bzw. Benachrichtigungen. Eine Subscription hat eine ihr zugewiesene Menge an überwachten Elementen. Welche Elemente einer Subscription zugewiesen sind, wird über das MonitoredItem Service Set definiert. Diese Subscriptions haben Sendeintervalle, die unabhängig von den Abtastintervallen der überwachten Elemente sind. Die Intervalle und weitere Einstellungen lassen mit Hilfe dieses Service Sets festlegen. [12, S. 72–86]

2.2 Web-API

API steht für *Application Programming Interface* (Anwendungs-Programmier-Schnittstelle) und stellt eine Programmierschnittstelle dar, welche die Interaktion zwischen verschiedenen Systemen oder Programmen ermöglicht bzw. vereinfacht. Die API bietet dabei einen Service, der über spezifizierte Schnittstellen bereitgestellt wird. Eine Abstraktion einer API stellt beispielsweise eine Steckdose dar. Diese bietet dem Konsumenten einen Service in Form eines Stromzugangs an. Die Steckdosenform stellt dabei die API dar, die ein Standard zur Ansteuerung des Stromnetzes liefert, welcher genau definiert ist durch Stromstärke sowie Spannung. Der Vorteil hierbei ist, dass sich der Konsument und die Hersteller von Programmen, welche die API nutzen, keine Gedanken darüber machen müssen, wie das System hinter der API funktioniert, weil dies durch die API bereitgestellt wird. [14, S. 2], [15]

Eine Web-API stellt dabei eine API für das Web dar, welche auf Web-Standards setzt.

2.3 HTTP

HTTP steht für Hypertext Transfer Protocol und ist ein Protokoll zur Übertragung von Daten im World Wide Web. Die erste Version HTTP/0.9 wurde im Jahr 1991 veröffentlicht, während die aktuelle Version HTTP/2 im Mai 2015 herausgegeben wurde. Das Protokoll arbeitet nach dem Request-Response-Prinzip, bei der ein Client eine Anfrage (Request) für eine Ressource an den Server sendet, auf die der Server eine Antwort (Response) sendet. Diese Antwort enthält Statusinformationen zu der Anfrage und kann auch die angefragte Ressource enthalten. [16]

Im Folgenden werden Beispiele für dieses Request-Response-Verfahren aufgezeigt und das Prinzip der HTTP-Statuscodes sowie der HTTP-Methoden erklärt.

2.3.1 Request-Response-Prinzip

2.3.1.1 URI

Jede Ressource im World Wide Web ist eindeutig über eine Id identifizierbar. Diese Id wird über eine *URI* (Universal Resource Identifier) beschrieben. Die Ressource sollte über diese URI dereferenzieren lassen, also ohne weitere Suche in einem Verzeichnis erreichbar sein, wobei das Verzeichnis in diesem Fall das Web darstellt. Eine URI ist dabei laut Spezifikation nicht auf den Einsatz für HTTP begrenzt. [17, S. 41–42]

Der Aufbau bzw. die Syntax einer URI ist dabei wie folgt:

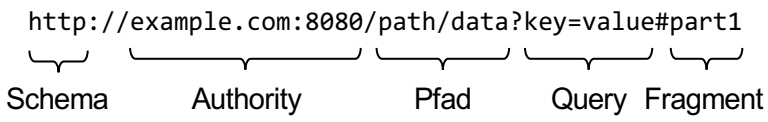


Abbildung 5: URI-Syntax (Quelle: eigene Darstellung in Anlehnung an [18])

Das Schema beschreibt den Kontext bzw. den Typ der URI, womit die Regeln der Interpretation des folgenden Teils festgelegt wird. *Authority* stellt den Hostnamen dar, was eine Domäne oder eine IP-Adresse sein kann, gefolgt von einem Port. Ein Pfad als solcher lässt sich als absoluter Pfad, wie in Abbildung 5 darstellen oder als relativer Pfad. Relative Pfade kann man zum Beispiel in der abgefragten Repräsentation einer Ressource finden, wo sie die relative Beziehung zwischen zwei Ressourcen abbilden. Die Query dient dazu Parameter zu übergeben, wobei im Beispiel aus Abbildung 5 der *key* den Namen des Parameters angibt und *value* dessen Wert ist. Weitere Parameter lassen sich durch ein Et-Zeichen *(&)*, oder auch Kaufmanns-Und genannt, hinzufügen. *Fragment* verweist auf einen bestimmten Teil einer zurückgegebenen Ressource. Im Falle einer Internetseite wird zum Beispiel das gesamte

Dokument übertragen und durch die Angabe des Fragments wird im Nachhinein clientseitig auf den Teil des Dokuments verwiesen, indem der Browser zu der entsprechenden Stelle des Dokuments springt. [17, S. 43–45], [18]

2.3.1.2 Request

Eine Request besteht aus einer Anfragezeile, einem *Header* (Kopfzeile) und einem *Body* (Rumpf). Die Anfragezeile beinhaltet die eingesetzte HTTP-Methode, die URI sowie die HTTP-Version. Auf die HTTP-Methoden wird im Kapitel 2.3.2 eingegangen. Nach dieser Zeile folgt der optionale Header, welcher es erlaubt, zusätzliche Informationen über die Anfrage selbst oder den Client, an den Server zu senden. Der Body enthält die Daten, welche an den Client gesendet werden sollen. Üblicherweise verfügen Anfragen mit HTTP-Methoden, die zum reinen Bezug von Daten dienen, nicht über einen solchen Body. [19], [20, Kap. 5]

In Listing 2 wird ein Beispiel für eine HTTP-Request gezeigt, welche auf die Benutzung von Header und Body verzichtet.

```
1  GET /path/data HTTP/1.1
```

Listing 2: Beispielhafte HTTP-Anfrage (Quelle: eigene Darstellung)

2.3.1.3 Response

Die Response (Antwort) besteht aus einer Statuszeile, einem Header und den Body. In der Statuszeile wird die Protokollversion und ein HTTP-Statuscode inklusive eines Statustexts zurückgegeben. Die Statuscodes sind dabei vordefinierte Codes, welche durch die *IETF* (Internet Engineering Task Force) spezifiziert sind. Der Header beinhaltet Informationen zur Response an sich und dem Server. Beispielsweise lassen sich im Header Informationen zum Datentyp und der eingesetzten Kodierung der im Body übermittelten Nachricht finden. [19]

In Listing 3 wird eine solche HTTP-Response dargestellt.

```
1  HTTP/1.1 200 OK
2  Server: Apache
3  Content-Type: text/html; charset=utf-8
4  Content-Lenght: 31
5  Date: Sun, 17 Oct 2021 13:24:13 GMT
6
7  <HTML>A simple HTML page</HTML>
```

Listing 3: Beispielhafte HTTP-Antwort (Quelle: eigene Darstellung)

2.3.2 HTTP-Methoden

Um eine uniforme Schnittstelle zu schaffen, wurden Operationen entworfen, welche auf alle Ressourcen eine festgelegte Auswirkung haben sollen. Insgesamt wurden dafür neun HTTP-Methoden definiert, welche im Folgenden kurz erklärt werden.

GET

Die GET-Methode fragt die Repräsentation der in der URI angegebenen Ressource an. GET sollte nur dazu genutzt werden, um Daten abzufragen und nicht um eine Zustandsänderung anzufordern. GET ist die am meisten eingesetzte HTTP-Methode mit geschätzten 95 % aller Interaktionen im World Wide Web. [17, S. 53–55], [21]

HEAD

Bei HEAD werden alle Metadaten abgefragt, welche auch bei einer GET-Anfrage über den Header übertragen werden, ohne jedoch den Body mitzusenden. Damit lässt sich zum Beispiel der Zeitpunkt der letzten Änderung einer Ressource abfragen. Dieser kann dazu genutzt werden, um zu prüfen, ob eine erneute Abfrage der Ressource notwendig ist oder ob die bereits in der Vergangenheit abgefragte Ressource noch auf den aktuellen Stand ist. [17, S. 55–56]

PUT

Die PUT-Methode wird dazu genutzt, um eine bestehende Ressource zu aktualisieren. Sollte diese Ressource nicht existieren, wird sie unter dem angegebenen Pfad erstellt. Der Inhalt der Ressource wird dabei über den Body gesendet. [17, S. 56]

POST

Die Methode wird zum einen dazu genutzt, um eine Ressource anzulegen und für alle Zwecke, welche sich nicht auf eine der anderen Methoden abbilden lassen. POST wird für den letzteren Zweck oft missbräuchlich benutzt, weil die HTTP-Spezifikationen keine Garantien festlegen, die die Methode erfüllen muss. Welche Garantien damit gemeint sind, wird am Ende dieses Kapitels erklärt. Beim Einsatz einer HTTP-POST-Methode zur Erstellung einer Ressource, wird nicht die URI der Ressource angesteuert, sondern die der für das Anlegen zuständigen Ressource, was häufig eine Listenressource darstellt, der die neue Ressource zugeordnet werden soll. [17, S. 57]

DELETE

DELETE wird zum Löschen der im Pfad angegebenen Ressource eingesetzt. Das Löschen muss dabei nicht zwingend zum Löschen von Daten auf dem Server führen. Da es sich bei den Ressourcen um Repräsentationen handelt, kann ein Löschen zum Beispiel auch das serverseitige Kennzeichnen einer Ressource als *storniert* sein, was für die Außendarstellung der Ressource den gleichen Effekt hat. [17, S. 57–58]

OPTIONS

Eine Anfrage über die OPTIONS-Methode liefert Informationen über die unterstützten HTTP-Methoden, die sich auf die Ressource anwenden lassen. [17, S. 58], [21]

TRACE

TRACE wird zur Fehleranalyse benutzt. Die Antwort sollte als Body die ursprüngliche Anfrage enthalten. Dies ermöglicht es zu sehen, welche Daten beim Empfänger angekommen sind. [17, S. 58]

CONNECT

Diese Methode wird zum Initiieren einer Ende-zu-Ende-Verbindung genutzt, was in Form von *HTTP-Tunneling* (Übersetzung und Übertragung in ein anderes Netzwerkprotokoll) stattfindet. Dies findet Anwendung beim Aufruf verschlüsselter Webseiten. [17, S. 58], [21]

PATCH

Im Gegensatz zur Methode PUT werden hierbei Instruktionen übermittelt, wie die Ressource verändert werden soll, anstatt die komplette Ressource zu senden. [19]

Die HTTP-Methoden bringen verschiedene Eigenschaften und Garantien mit sich, welche sie erfüllen müssen. Methoden können dabei zum einen *safe* (sicher) sein, was bedeutet das sie den Zustand einer Ressource nicht verändern [22]. Alle Methoden die safe sind, sind immer auch *idempotent*, was bedeutet das es egal ist, wie oft die gleiche HTTP-Anfrage ausgeführt wird, da sie immer zum gleichen Ergebnis führt [22], [23]. Eine weitere Eigenschaft ist die Cache-Fähigkeit der Methoden, was die Möglichkeit darstellt, die erhaltene Darstellung einer Ressource für die spätere Verwendung zwischenzuspeichern. Durch den Einsatz, wie bei der

HTTP-Methode HEAD beschrieben, können somit zukünftige GET-Anfragen vermieden werden [24]. In Tabelle 3 werden die Eigenschaften der einzelnen Methoden aufgeführt, wobei ein *X* dafür steht, dass die Eigenschaft erfüllt wird und ein *O* bedeutet, dass es möglich ist, aber nicht zwingend so sein muss.

Tabelle 3: Eigenschaften der HTTP-Methoden (Quelle: eigene Darstellung in Anlehnung an [17, S. 59], [21])

Methode	Safe	Idempotent	Request-Body	Response-Body	Cache-fähig	Identifizierbare Ressource
GET	X	X		X	X	X
HEAD	X	X			X	X
PUT		X	X			X
POST			X	X		
DELETE		X	O	O		X
OPTIONS	X	X		X		X
TRACE	X	X				X
CONNECT				X		X
PATCH			X	X		X

2.3.3 HTTP-Statuscodes

Der HTTP-Statuscode wird in der Antwort einer HTTP-Anfrage mitgeliefert. Er gibt Auskunft über den Erfolg bzw. Misserfolg einer Anfrage. Durch die Nutzung der HTTP-Statuscodes ist ein vereinheitlichtes Fehlermanagement durch alle Nutzer möglich, die die HTTP-Spezifikation unterstützen. Unterteilt werden die Statuscodes dabei in fünf Gruppen, welche im Folgenden anhand einer Auswahl von Statuscodes erläutert werden. [25], [26]

Informative Antworten (1xx)

Diese Statuscodes enthalten Informationen wie zum Beispiel, dass die Bearbeitung noch andauert *(102 Processing)* oder die Protokolle gewechselt werden *(101 Switching Protocols)*. Letzteres wird unter anderem beim Aufbau von WebSocket-Verbindungen genutzt. [25], [26]

Erfolgreiche Antworten (2xx)

Diese Statuscodes zeigen an, dass die Anfrage erfolgreich war. Beispiele hierfür sind der Statuscode *200 OK*, welcher mitunter bei den HTTP-Methoden GET, HEAD und POST als Antwort geliefert werden kann sowie der Statuscode *201 Created* bei einer erfolgreichen PUT-Anfrage. [25], [26]

Umleitung (3xx)

Bekannte Vertreter dieser Gruppe sind der Code *301 Moved Permanently*, welcher anzeigt, dass die Ressource unter einer neuen Adresse verfügbar ist. Der Statuscode *304 Not Modified* gibt an, dass sich eine Ressource seit der letzten Abfrage nicht geändert hat. [25], [26]

Client-Fehler (4xx)

Diese Fehler zeigen ein Scheitern der Anfrage an, wobei der Verantwortungsbereich auf der Seite des Clients liegt. Der Statuscode *400 Bad Request* deutet zum Beispiel auf eine fehlerhafte Anfrage-Nachricht hin, während *404 Not Found* auf eine nicht vorhandene Ressource hinweist. Ein weiteres Beispiel ist *403 Forbidden*, was bedeutet, dass der Client nicht über die benötigten Rechte verfügt, um auf die Ressource zuzugreifen. [25], [26]

Server-Fehler (5xx)

Die Ursache dieser Statuscodes liegt eher im Verantwortungsbereich des Servers, lässt sich jedoch nicht klar von Client-Fehlern abgrenzen. Ein aus dieser Gruppe häufig auftretender Fehler ist *500 Internal Server Error*, welcher ein Sammel-Statuscode für unerwartete Serverfehler ist. Sollte eine Funktionalität serverseitig noch nicht implementiert sein, wird dies durch ein *501 Not Implemented* beantwortet. [25], [26]

3 Systematische Analyse

In diesem Teil der Arbeit wird der Soll- und Ist-Zustand analysiert. Aus dieser Analyse werden die Anwendungsfälle des Projektes gebildet sowie die funktionalen und die nicht-funktionalen Anforderungen.

3.1 Istzustand

Trotz dessen, dass OPC UA so entworfen wurde, dass es unabhängig von dem eingesetzten Transportprotokoll ist, werden zwei Protokolle in den Spezifikationen angegeben [1, S. 7]. TCP ist eines davon und stellt eine Ende-zu-Ende-Verbindung dar, welche vollduplex-fähig ist. Das bedeutet, dass die Kommunikation in beide Richtungen möglich ist. [2, S. 59], [27]

TCP sendet dabei die Daten in einzelnen Paketen, deren Eintreffen durch den Empfänger bestätigt wird. Dies ermöglicht eine verlustfreie Datenübertragung. Neben diesem Protokoll bietet OPC UA auch eine Unterstützung für *HTTPS*, was eine verschlüsselte HTTP-Kommunikation über das Protokoll TLS ist. Die Syntax der Nachrichten ändert sich durch den Einsatz von HTTP nicht. Es wird lediglich eine abgesicherte Verbindung zur Übertragung der Daten genutzt. Als Transportschicht findet weiterhin TCP Anwendung. [2, S. 59]

In der Praxis wird der Einsatz von TCP bevorzugt, da HTTPS eine schlechtere Performance bietet und die Implementierung des Protokolls nicht sehr verbreitet ist. [28, S. 13], [29, S. 9]

3.2 Sollzustand

Es wird eine Web-API bereitgestellt, die zur Kommunikation mit einem OPC UA Server einen OPC UA Client als Schnittstelle nutzt und auf moderne Webtechnologien setzt, um die Daten zur Verfügung zu stellen. Die gewonnenen Informationen werden in Form von einer Datenkodierung wie JSON oder XML ausgegeben. Diese Datenkodierungen sind zwar nicht so performant wie die durch OPC UA unterstützten binären Daten, bieten dafür aber ein durch Menschen lesbares Format, welches mit vielen Programmiersprachen verarbeitet werden kann [30, S. 31], [31].

3.3 Anwendungsfallanalyse

An dieser Stelle werden die einzelnen Anwendungsfälle (use cases) nach den Empfehlungen des Buches *Lehrbuch der Software-Technik* von *Helmut Balzert*[4] als Use-Case-Diagramm modelliert. Dabei werden die einzelnen Akteure definiert, welche mit der Software kommunizieren oder Daten austauschen und somit einen gewissen Einfluss auf das System nehmen. Diese Einflussnahme sind Arbeitsabläufe, die mit der Software durchgeführt werden und welche im weiteren Verlauf als Anwendungsfälle bezeichnet werden. Die Anwendungsfälle richten sich hierbei nach Operationen, die sich in Verbindung mit einem OPC UA Server durchführen lassen, unabhängig vom Transportszenario oder der technischen Umsetzbarkeit. Diese Operationen stellen die Verwendung der OPC UA Services dar, wobei jedoch im Folgenden eine Einschränkung der durch diese Web-API unterstützten Services vorgenommen wird. Dies hat den Hintergrund, dass die Web-API vor allem zur Bedienung von Anlagen bzw. Maschinen gedacht ist, welche bereits fertig konfiguriert sind. Wichtig dabei ist es, Daten abzufragen und an den Server zu senden.

Das NodeManagement Service Set, welches zum Hinzufügen und Löschen von Nodes bzw. Referenzen dient, stellt hingegen eher eine Konfiguration des Servers dar und kein klassisches Bedienverhalten. Aus diesem Grund wird von einer Implementierung dieses Service Sets abgesehen.

Ein weiterer Service, von dessen Implementierung in dieser Web-API abgesehen wird, ist der Service HistoryUpdate aus dem Attribute Service Set. Historische Werte dienen zur Nachverfolgbarkeit von Werten, die aufgezeichnet wurden. Eine Änderung dieser Werte ist zwar durch den OPC UA Service möglich, jedoch nicht in der Praxis vorgesehen, da dies eine Manipulation von Werten darstellen würde, welche eigentlich nicht manipuliert werden sollen. Untermauert wird dies auch durch das Fehlen der Funktion in bekannten Clients wie dem UaExpert von Unified Automation und dem OPC UA Browser von Prosys [33], [34].

In Abbildung 6 wurden diese Anwendungsfälle mit Hilfe eines Use-Case-Diagramms zusammengefasst.

[4] [32, S. 65]

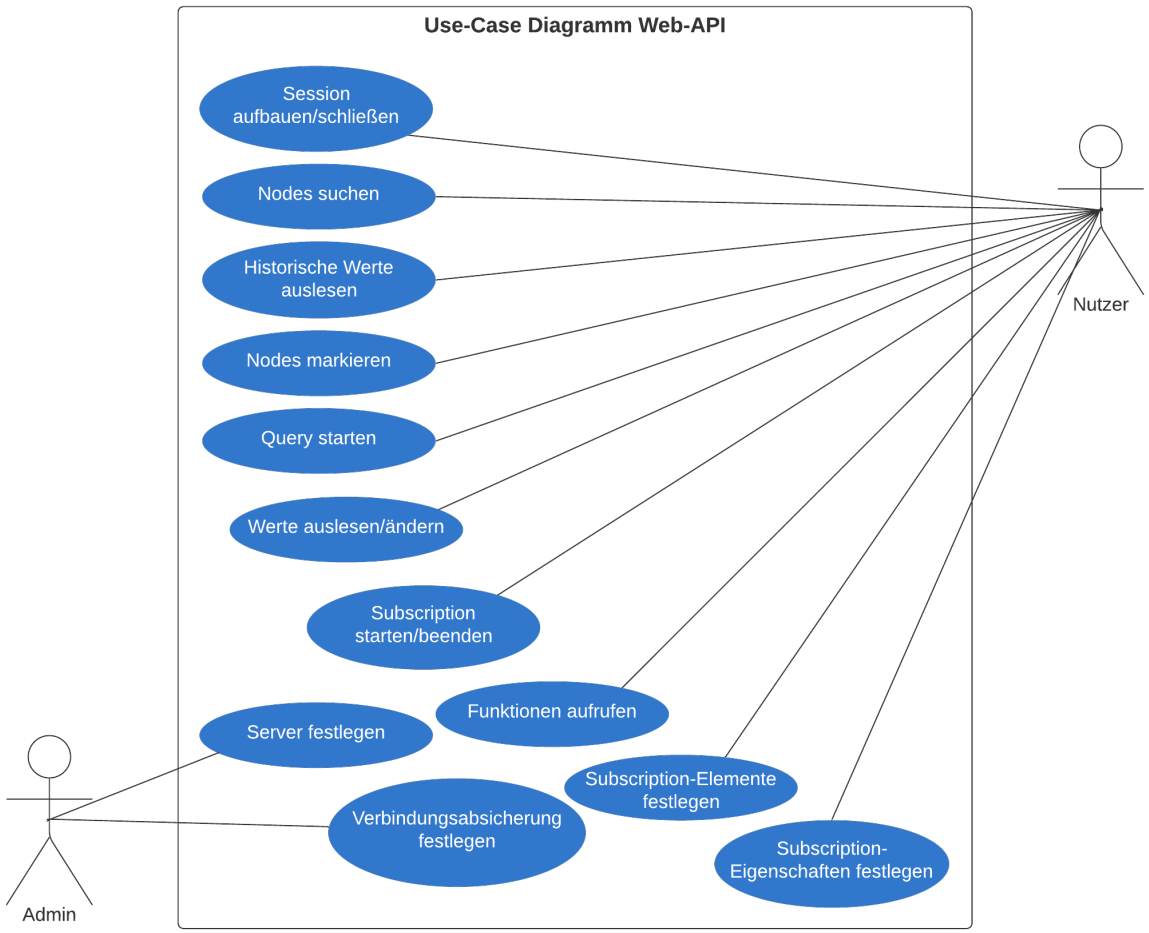


Abbildung 6: Use-Case-Diagramm für die Web-API (Quelle: eigene Darstellung)

3.4 Anforderungsanalyse

Aus den in Abbildung 6 genannten Anwendungsfällen lassen sich Anforderungen an das System formulieren. Anforderungen können sich dabei direkt oder indirekt aus den Anwendungsfällen ergeben. Die Anforderungen werden in diesem Abschnitt in funktional und nicht-funktional unterteilt. Sie werden mit einer laufenden Nummer angegeben, wobei *FA* für die funktionalen Anforderungen und *NFA* für die nicht-funktionalen Anforderungen steht. Die einzelnen Anforderungen werden mit einer Priorität versehen, welche als Orientierungshilfe bei der Vorgehensweise in der Implementierungsphase dient. Weiterhin wird ein Bezug auf die vorher genannten Anwendungsfälle angegeben.

3.4.1 Funktionale Anforderungen

Die funktionalen Anforderungen haben einen Bezug zur Zweckbestimmung des Produkts. Sie bilden damit direkt die Funktionen ab, welche das Produkt bieten soll. [32, S. 115–116]

FA_01

Titel: Festlegung Server

Beschreibung: Es muss die Funktion geboten werden, den Server für den Verbindungsaufbau festzulegen.

Bezug: Grundvoraussetzung für alle Anwendungsfälle

Priorität: hoch

FA_02

Titel: Verbindungsaufbau OPC UA Server

Beschreibung: Die Möglichkeit der Verbindung über die API zu einem vorher definierten OPC UA Server muss gegeben sein.

Bezug: Grundvoraussetzung für alle Anwendungsfälle

Priorität: hoch

FA_03

Titel: Zugriff über das Web

Beschreibung: Die API muss über eine Webschnittstelle erreichbar sein.

Bezug: Grundvoraussetzung für alle Anwendungsfälle

Priorität: hoch

FA_04

Titel: Session aufbauen

Beschreibung: Mit vorher festgelegten Sicherheitsmaßnahmen muss es möglich sein, eine Session mit dem OPC UA Server aufzubauen und zu aktivieren.

Bezug: Session aufbauen/schließen

Priorität: hoch

FA_05

Titel: Session schließen

Beschreibung: Die Software soll die Funktion zum Schließen bzw. zum Beenden einer Session bereitstellen, bei der auch alle noch anstehenden Operationen der Session beendet werden.

Bezug: Session aufbauen/schließen

Priorität: hoch

FA_06

Titel: Adressraum durchsuchen

Beschreibung: Die Software ermöglicht es den Adressraum zu durchsuchen und somit die Nodes und deren Beziehungen zueinander zu erfassen.

Bezug: Nodes suchen

Priorität: hoch

FA_07

Titel: Nodes markieren

Beschreibung: Nodes sollen in einer Form markierbar sein, wodurch dem OPC UA Server signalisiert werden kann, dass oft auf diese Nodes zugegriffen wird, damit der Server die Performance für den Zugang zu diesen Daten optimieren kann.

Bezug: Nodes markieren

Priorität: niedrig

FA_08

Titel: Attribute lesen

Beschreibung: Es ist möglich die aktuellen Werte von Attributen einer Node zu lesen.

Bezug: Werte auslesen/ändern

Priorität: hoch

FA_09

Titel: Werte schreiben

Beschreibung: Es wird die Möglichkeit geboten Werte von Attributen einer Node zu schreiben bzw. an den OPC UA Server zu senden.

Bezug: Werte auslesen/ändern

Priorität: hoch

FA_10

Titel: Historische Werte lesen

Beschreibung: Neben dem Lesen der aktuellen Werte soll die Software auch die Möglichkeit bieten, unter Angabe eines Zeitraums die aufgezeichneten Werte aus der Vergangenheit eines Attributes zu lesen.

Bezug: Historische Werte auslesen

Priorität: hoch

FA_11

Titel: Query starten

Beschreibung: Die Software muss die Funktion bieten eine Query zu starten, mit der ein umfassender Zugriff auf Daten möglich ist, wobei die Auswahl der Daten über einen Filter stattfindet.

Bezug: Query starten

Priorität: hoch

FA_12

Titel: Funktionen aufrufen

Beschreibung: Der Nutzer soll über die Software dazu in der Lage sein, Funktionen auf dem OPC UA Server aufzurufen. Es soll weiterhin möglich sein, der Funktion Werte mitzugeben und Werte von ihr zu erhalten.

Bezug: Funktionen aufrufen

Priorität: hoch

FA_13

Titel: Subscription starten

Beschreibung: Die Software soll dazu in der Lage sein eine Subscription zu starten, wodurch die Software aktuelle Werte von vorher definierten Attributen an den Nutzer sendet. Ein erneutes Senden der Daten findet bei jeder Änderung der Daten bzw. bei neuen Daten statt.

Bezug: Subscription starten/beenden

Priorität: hoch

FA_14

Titel: Subscription beenden

Beschreibung: Die Software soll dazu in der Lage sein, eine Subscription zu beenden.

Bezug: Subscription starten/beenden

Priorität: hoch

FA_15

Titel: Subscription-Elemente festlegen

Beschreibung: Der Nutzer ist durch die Software dazu in der Lage festzulegen, welche Elemente durch eine Subscription überwacht werden sollen.

Bezug: Subscription-Elemente festlegen

Priorität: hoch

FA_16

Titel: Subscription-Eigenschaften festlegen

Beschreibung: Durch die Software wird es dem Nutzer ermöglicht, die Eigenschaften der Subscription zu ändern.

Bezug: Subscription-Eigenschaften festlegen

Priorität: niedrig

FA_17

Titel: Moderne Datentypen

Beschreibung: Bei dem Datentyp der Antwort auf die HTTP-Anfrage soll auf moderne Datentypen wie beispielsweise XML oder JSON gesetzt werden.

Bezug: Grundvoraussetzung für alle Anwendungsfälle

Priorität: hoch

3.4.2 Nicht-funktionale Anforderungen

Dies sind jene Anforderungen, die sich nicht direkt auf die Funktionalität bzw. Leistung des Produktes beziehen. Beispiele hierfür sind Sicherheitsanforderungen oder Anforderungen, die mit der Revisionsfähigkeit in Beziehung stehen. [32, S. 117]

NFA_01

Titel: Verbindungsabsicherung

Beschreibung: Die Verbindung muss die Möglichkeit einer Absicherung bieten, um die Integrität und Vertraulichkeit der Kommunikation zu bewahren. Integrität ist dabei der Schutz vor unbemerkten Änderungen durch Dritte und Vertraulichkeit bedeutet, dass die Daten nur für authentifizierte Parteien offengelegt werden.

Bezug: Verbindungsabsicherung festlegen

Priorität: mittel

NFA_02

Titel: Dokumentation

Beschreibung: Die API soll eine Dokumentation für die Implementierung durch Dritte besitzen. Diese kann im Quellcode oder durch entsprechende Dokumentationssoftware erfolgen.

Priorität: mittel

NFA_03

Titel: Lizenzfreie Software

Beschreibung: Um die Nutzung durch Dritte zu vereinfachen, soll die Software keine kostenpflichtigen Plug-Ins, SDKs oder Frameworks nutzen.

Priorität: hoch

NFA_04

Titel: Standards & Frameworks

Beschreibung: Für die Umsetzung der Web-API soll auf bekannte und erprobte Standards, SDKs und Frameworks gesetzt werden.

Priorität: hoch

NFA_05

Titel: Code-Qualität

Beschreibung: Der Code soll auf eine verbreitete Programmiersprache setzen und deren offiziellen Sprachkern nutzen. Der Code muss leicht zu verstehen sein, damit spätere Anpassungen durch andere Personen problemlos möglich sind.

Priorität: hoch

NFA_06

Titel: Testbarkeit

Beschreibung: Die Testbarkeit der Software soll bspw. durch Angabe von Testszenarien gewährleistet sein.

Priorität: mittel

NFA_07

Titel: Web-Affinität

Beschreibung: Die Software soll auf web-affine Technologien, also auf gängige Standards im Internet setzen und deren Protokolle unterstützen sowie nutzen.

Priorität: hoch

4 Konzeption

In diesem Abschnitt wird zu Anfang auf die Auswahl von Hilfsmitteln, wie die Programmiersprache, SDKs und Frameworks eingegangen. Anschließend folgt eine Unterteilung der zur Auswahl stehenden Technologien zur Datenbeschaffung nach Web-APIs und *Server-Push-Technologien*, wobei diese genauer untersucht werden. Klassische Web-APIs arbeiten vorwiegend nach dem Request-Response-Paradigma. Bei Push-Technologien hingegen übernimmt der Server die Überwachung der Daten und sendet die Daten bei Feststellung einer Änderung oder nach einem bestimmten Zeitintervall an den Client.

Die Einteilung in diese zwei Gruppen von Technologien ist dem Umstand geschuldet, dass OPC UA nach einem ähnlichen Prinzip arbeitet, bei dem die Server-Push-Technologie die Subscriptions abbilden soll und die Web-API die restlichen Services, die nach dem Request-Response-Prinzip arbeiten.

4.1 Auswahl der Hilfsmittel

In diesem Kapitel erfolgt eine Auswahl von Hilfsmitteln wie der Programmiersprache, Frameworks und damit verbundenen weiteren Hilfsmitteln. Bei der Auswahl dieser Hilfsmittel ist oft die Verbreitung entscheidend, da diese ein Indikator dafür ist, wie geeignet die Mittel bei der Umsetzung von Projekten sind. Bekannte Hilfsmittel vermeiden die Probleme unbekannter bzw. weniger verbreiteter Hilfsmittel, wie fehlende Weiterentwicklung bzw. Wartung, eine lückenhafte Dokumentation, kaum vorhandene technische Unterstützung oder fehlende Unterstützung durch andere Bibliotheken.

4.1.1 Programmiersprache

JavaScript ist eine objektorientierte Programmiersprache, die sich besonders zur Entwicklung im Web eignet und laut einer Umfrage im neunten Jahr in Folge die meistgenutzte Programmiersprache bei professionellen Entwicklern ist [35], [36]. Des Weiteren ist JSON, eines der von OPC UA unterstützten Formate zum Datenaustausch, von JavaScript abgeleitet und nutzt die Syntax von JavaScript-Objekten. Diese beiden Faktoren und die einfache Erlernbarkeit von JavaScript, sind ausschlaggebend für die Wahl. Bei der Umsetzung wird auf den 2015 veröffentlichten Sprachkern *ECMAScript 6* gesetzt, welcher durch eine Vielzahl aktueller Browser und Server beinahe komplett unterstützt wird [37]. Durch die hier getroffene Auswahl wird ein Teil der nicht-funktionalen Anforderung NFA_05 erfüllt.

4.1.2 SDK

NodeOPCUA ist ein auf der Laufzeitumgebung *Node.js* basierendes SDK für OPC UA, dessen Entwickler mit dieser SDK als ein Mitglied der OPC UA Foundation gelistet ist [38]. NodeOPCUA kann mit JavaScript oder TypeScript genutzt werden und erfüllt somit die Anforderungen an die gewählte Programmiersprache. Weiterhin ist diese SDK frei zugänglich, Modifikationen sind möglich und sie genießt eine breite Unterstützung durch große Firmen wie IBM, SAP, Siemens und vielen weiteren, wodurch die Voraussetzungen der nicht-funktionalen Anforderungen NFA_03 und NFA_04 abgedeckt werden. [39]

4.1.3 Webframework

Express.js dient dazu festzulegen, welche Funktionen beim Aufruf von bestimmten URIs über die verschiedenen HTTP-Verben ausgeführt werden sollen. Express ist eines der meistgenutzten Module für Node.js und wird von vielen großen Firmen unterstützt [40], [41]. Durch die Auswahl eines so bekannten und etablierten Frameworks werden die Bedingung aus den nicht-funktionalen Anforderung NFA_03 und NFA_04 erfüllt.

4.1.4 Entwicklungsumgebung

Bei der Entwicklungsumgebung wird auf eine *IDE* (Integrated Development Environment) gesetzt. Dies ist eine Entwicklungsumgebung, die im Gegensatz zu einem einfachen Texteditor auch weitere Programmierwerkzeuge zur Verfügung stellt, wie zum Beispiel einen *Compiler* zur Übersetzung in Maschinen-Code oder einen *Debugger* zur Fehlerdiagnose, welche bei der Entwicklung hilfreich sein können [42].

Nach dem *State of JS Report* von 2020[5], bei dem ca. 23000 Personen befragt wurden, sind *Visual Studio Code*, *Vim* und *WebStorm* die drei am meisten genutzten Editoren zur Programmierung. Da die oft genutzten IDEs sich nur unwesentlich im Funktionsumfang unterscheiden und alle ihre Vor- sowie Nachteile bieten, fällt die Wahl auf Grund persönlicher Erfahrungen mit der IDE auf WebStorm.

[5] [43]

4.1.5 API-Dokumentation

Da dieses Projekt, wie in Kapitel 1.4 beschrieben, kein vollständig implementiertes Projekt vorlegt und sich der Einsatz vorerst auf die Hochschule Merseburg konzentriert, wird auf eine ausführliche Dokumentation innerhalb des Quelltextes gesetzt. Dies ist für den vorgesehenen Einsatzzweck vorerst ausreichend und erfüllt die Anforderung NFA_02. Über die Angabe von Testszenarien wird außerdem die Möglichkeit geboten den Code zu testen (NFA_06).

4.2 Auswahl der Request/Response-Technologie

Eine der ersten großen Web-APIs stellte *SOAP* (Simple Object Access Protocol) dar. SOAP wurde im Jahr 1999 entwickelt und ist ein Protokoll zum Austausch von Nachrichten auf der Basis von XML [44]. Die gegebenen Restriktionen, der Aufwand der Umsetzung und die gleichzeitig vorhandene Offenheit bei der Auswahl der Transportprotokolle sorgten dafür, dass SOAP in der heutigen Zeit nur noch selten Anwendung findet, außer bei großen Unternehmen, deren Systeme auf diesen Restriktionen basieren [14, S. 10], [17, S. 10].

Im Jahr 2000 wurde der *REST*-Architekturstil (Representational State Transfer) in Form einer Dissertation veröffentlicht. Im Laufe der Jahre hat das Interesse an RESTful-Webservices und die Verbreitung immer weiter zugenommen, was unter anderem auch daran lag, dass es einen großen Interpretationsspielraum und damit auch viele Möglichkeiten für die Umsetzung dieses Architekturstils gab. Das Interesse an SOAP sank und Implementierungen von SOAP wurden in vielen Bereichen durch RESTful-Webservices ersetzt. [14, S. 10], [17, S. 10], [45], [46]

Zwölf Jahre nach der Veröffentlichung von REST gab es erste Bestrebungen neue Technologien zu entwickeln, welche mit REST konkurrieren konnten. Facebook begann intern mit der Entwicklung von *GraphQL* im Jahr 2012 und veröffentlichte es 2015. Seit 2018 wird das Projekt durch die Linux Foundation betreut. [47]

Noch im selben Jahr wurden mit *gRPC* und *JSON:API* zwei weitere Web-APIs veröffentlicht. Bei gRPC handelt es sich um ein durch Google entwickeltes Framework, welches unter der Apache Lizenz 2.0 veröffentlicht wurde und auf die erweiterten Funktionen von HTTP/2 setzt, während JSON:API eine freie Spezifikation ist, die eher als Erweiterung von REST angesehen werden kann. [14, S. 20], [47], [48]

Im weiteren Verlauf dieses Kapitels werden Erklärungen der einzelnen Technologien sowie ein Vergleich anhand ihrer Stärken und Schwächen durchgeführt. Mittels dieses Vergleichs wird am Ende eine passende Technologie für das Projekt ausgewählt.

4.2.1 Vergleich

Im Folgenden werden die APIs kurz vorgestellt und ein Überblick über die Funktionsweise sowie deren Vor- und Nachteile gegeben. Auf das Netzwerkprotokoll SOAP wird bei dieser Betrachtung verzichtet. In dieser Arbeit wird nach einer passenden Web-API für den Zugriff auf OPC UA gesucht. Das SOAP aus neueren Versionen von OPC UA auf Grund fehlender Akzeptanz seitens der Industrie gestrichen wurde zeigt, dass es sich dabei nicht um ein geeignetes Protokoll für die Umsetzung dieser Aufgabe handelt [2, S. 45].

4.2.1.1 REST

Wie bereits erwähnt, handelt es sich bei REST um ein Architekturstil bzw. um ein Programmierparadigma. Dies führt dazu, dass es viel Interpretationsspielraum bei der Umsetzung des Paradigmas gibt. Bei REST werden Daten als Ressourcen gesehen, die über eine eindeutige URI adressiert werden können. Die Kommunikation findet dabei mittels der HTTP-Methoden statt, wobei die vier CRUD-Methoden (Create, Read, Update und Delete) verwendet werden. Die Kommunikation funktioniert nach dem Request-Response-Schema. [14, S. 10–12]

Das Prinzip von REST lässt sich dabei auf folgende fünf Punkte reduzieren [17, S. 11]:

Eindeutig identifizierbare Ressourcen

Alles was sich als Ressource abstrahieren lässt, soll einer eindeutigen URI zugewiesen sein [17, S. 12]. Die folgenden URIs bilden beispielhaft eine einzelne Messung, eine Liste von Messungen und eine Methode zum Starten einer Umpump-Automatik ab.

- *http://example.com/synthese/sensoren/temperatur/TI3056*
- *http://example.com/synthese/sensoren/temperatur*
- *http://example.com/destillation/umpump-automatik-ein*

Wie in diesem Beispiel zu sehen ist, stellen alle durch einen Schrägstrich getrennten Begriffe eine Ressource dar, egal ob diese nun einzelne Elemente sein sollen oder eine Menge von Elementen.

HATEOAS

Hypermedia as the engine of application state auch HATEOAS genannt, ist ein Konzept, bei dem in der Repräsentation einer Ressource Verbindungen zu anderen Ressourcen bereitgestellt werden. Das ermöglicht es einen Client, alle Endpunkte dynamisch zu erkunden und durch den Adressraum zu navigieren. Dies wird durch das Beispiel einer Response im Listing 4 veranschaulicht, bei dem der Temperatursensor und das Ventil als Verlinkungen hinterlegt sind. [14, S. 12], [17, S. 13–14]

```
{
  "type": "Temperaturregelung",
  "name": "TC3056",
  "sensor": {
    "href": "http://example.com/synthese/sensoren/temperatur/TI3056"
  },
  "valve": {
    "href": "http://example.com/synthese/ventile/4T015051014"
  }
}
```

Listing 4: Beispiel-Code für HATEOAS (Quelle: eigene Darstellung)

Standardmethoden

REST setzt üblicherweise auf vier der Standardmethoden, die HTTP mit sich bringt. Diese Methoden bilden die vier grundlegenden Operationen persistenter, also dauerhafter Speicher ab. In Tabelle 4 werden die einzelnen HTTP-Methoden den Operationen zugeordnet. [17, S. 14–17], [49]

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

Operation	HTTP-Verb	Auswirkung
Create	POST	Anlegen einer Ressource
Read	GET	Anfordern einer Ressource
Update	PUT	Ändert eine Ressource bzw. legt diese an, wenn sie noch nicht vorhanden ist.
Delete	DELETE	Löschen einer Ressource

Der Vorteil hierbei ist, dass diese Methoden allgemein bekannt bzw. definiert sind und damit die in den Grundlagen zu HTTP genannten Eigenschaften mit sich bringen. Dies wirkt zwar wie eine vermeintliche Einschränkung bei der Funktionsgestaltung, sorgt aber dafür, dass der Client nicht zuerst mit allen Methoden vertraut gemacht werden muss, da diese allgemein bekannt sind. [17, S. 14–17]

Ressourcen und Repräsentationen

Ressourcen sollten in verschiedenen Formaten zur Verfügung gestellt werden, damit der Client diese auch verarbeiten kann. [17, S. 17–18]

Statuslose Kommunikation

Ein Server soll sich für den Status des Clients nur für die Zeit der Abarbeitung der Anfrage interessieren. Alle Änderungen, die persistent sein sollen, müssen auf dem Server in einen Ressourcenstatus umgewandelt werden oder als Teil der Repräsentation an den Client übermittelt werden. Jede erneute Anfrage des gleichen Clients wird also wie eine Anfrage eines neuen Clients behandelt. [17, S. 18–19]

Durch das Code-Beispiel in Listing 5 wird illustriert, wie eine Anfrage einer Ressource und die Antwort unter REST aussehen könnten. Dieses Beispiel dient im Folgenden zur Veranschaulichung der Unterschiede zwischen den einzelnen Technologien.

```
GET /api/tcs HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "data": [{
    "id": "TC3056",
    "kp": 1.877,
    "tn": 189,
    "tv": 16.5,
    "controllerId": "4T015051014",
    "meterId": "TI3056"
  }]
}
```

Listing 5: Code-Beispiel Request/Response mit REST (Quelle: eigene Darstellung)

Es ergeben sich einige Vorteile, die aus der Verwendung von REST resultieren. Durch den Einsatz von HTTP, welches Statuscodes anbietet, ist bereits ein weitverbreitetes und damit gut interpretierbares *Error-Handling* (Fehlerbehandlung) vorhanden. [50]

REST ist nicht an bestimmte Repräsentationsformate gebunden, wie es zum Beispiel bei SOAP mit XML der Fall ist. Ein häufiges eingesetztes Format ist zwar JSON, jedoch ist es eine reine Entwurfsentscheidung, welches Format zum Einsatz kommt. Mittels *Content Negotiation* kann vereinbart werden, welche Formate der Client bzw. Server akzeptiert. [17, S. 88]

Da das REST-Paradigma sehr generell formuliert wurde, gibt es einen großen Interpretationsspielraum bei der Auslegung und Umsetzung dieses Stils. Dies sorgt für eine hohe Flexibilität bei der Implementierung und den Einsatzszenarien.

Eine Autorisierung lässt sich bei REST gut umsetzen, da jede Ressource über einen Endpunkt abgebildet wird, womit über Authentifizierung leicht der Zugriff auf die einzelnen Endpunkte getrennt geregelt werden kann. [14, S. 93–94]

Ein weiterer Vorteil ergibt sich daraus, dass REST die Möglichkeit des Caching bietet. Das bedeutet, dass unnötige Anfragen von Daten bzw. die unnötige Übertragung dieser vermieden werden. Dies kann nach dem Expirationsmodell erfolgen, bei dem der Server in der Response angibt, für wie lange keine neue Anfrage der Ressource notwendig ist oder über das Validierungsmodell, bei dem im Header der HTTP-Anfrage ein Hash-Wert der vorhandenen Ressource gesendet wird, welcher von dem Server mit dem Hash-Wert der Server-Ressource abgeglichen wird. Wenn die Ressource unverändert ist, wäre die Antwort durch den Client mit dem HTTP-Statuscode *304 Not Modified* versehen. Beide Modelle können auch miteinander kombiniert werden, um die Anfragen beim Validierungsmodell zu reduzieren. [17, S. 127–131]

Die genannten Vorteile bringen jedoch auch Nachteile mit sich. Durch den Einsatz von HTTP sind keine Subscriptions möglich, da das Protokoll keine Funktion dafür vorgesehen hat. Eine Umsetzung von Subscriptions muss daher eigenständig entwickelt werden, sodass mögliche Implementierungen nicht nach einem einheitlichen Standard arbeiten.

Bei REST handelt es sich am Ende nur um einen Architekturstil, der als Dissertation einer einzelnen Person veröffentlicht wurde und nicht um eine Spezifikation wie bei den anderen im weiteren Verlauf vorgestellten Technologien. Der damit gegebene Interpretationsspielraum, der soeben noch als Vorteil von REST genannt wurde, kann bei schlechter Umsetzung dieses Paradigmas zu Implementierungen führen, die alles andere als RESTful sind [17, S. 10].

Ein weiteres großes Problem stellt das sogenannte *Over-* und *Under-fetching* dar. Zur Veranschaulichung wird das Beispiel eines Anlagenteils in einer Chemieanlage verwendet, der eine Menge von Ventilen verwaltet. Wenn nun lediglich der aktuelle Stellwert eines

bestimmten Ventils von Interesse ist, muss zuerst eine Anfrage gestartet werden, bei der eine Liste aller Anlagenteile zurückgegeben wird. Anschließend muss eine weitere Anfrage an die Ressource gesendet werden, die den gewollten Anlagenteil repräsentiert. Diese gibt dann Eigenschaften des Anlagenteils zurück sowie eine verlinkte Listressource aller Ventile. Nach einem Aufruf dieser Listenressource muss erneut eine Anfrage für das gesuchte Ventil gestellt werden, wobei die Werte und Eigenschaften des Ventils zurückgegeben werden, die neben dem Stellwert zum Beispiel auch eine Seriennummer und weitere Daten enthält, die in dem Moment nicht relevant sind. Die zurückgelieferten Daten, die bezogen auf die Anfrage irrelevant sind, beschreiben das Over-fetching. Zur gleichen Zeit erhält man jedoch auch zu wenige Daten, was als Under-fetching bezeichnet wird, da vier Anfragen gestellt werden müssen, wobei die ersten drei Anfragen die geforderten Daten nicht enthielten, also zu wenig Daten auswiesen. Dies sorgt für eine höhere Auslastung des Netzwerks, da zum einen zu viele Daten gesendet werden und mehr Anfragen als nötig gestellt werden müssen.

4.2.1.2 JSON:API

Die JSON:API ähnelt REST in einigen Punkten, wurde jedoch um ein Schema erweitert, welches die Beziehungen der Ressource zu anderen Ressourcen abbildet sowie weitere festgelegte Informationen. Über diese Beziehungen ist es möglich, sogenannte *Compound Documents* in die Antwort auf eine Anfrage zu inkludieren. Damit ist es möglich zusätzliche Anfragen zu vermeiden, wie es bei REST der Fall ist. Wie diese Compound Documents funktionieren, wird im Listing 6 aufgezeigt. [14, S. 12–15]

In diesem Code-Beispiel kann man sehen, dass das Element *Data* die Attribute der Ressource enthält. Über die *Relationships* werden die verknüpften Ressourcen angegeben. Mit Hilfe des Anfrage-Parameters *include=meter* kann der Client angeben, dass er zusätzlich an den Daten des Temperatursensors interessiert ist. Somit kann der Prozess, der bei REST aus zwei Anfragen bestanden hätte, über einer Anfrage gelöst werden.[14, S. 12–15], [51]

```
GET /api/tcs?include=meter HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [{
    "type": "TC",
    "id": "TC3056",
    "attributes": {
      "kp": 1.877,
      "tn": 189,
      "tv": 16.5
    },
    "relationships": {
      "controller": {"data": {"id": "4T015051014", "type": "valves"},
      "meter": {"data": {"id": "TI3056", "type": "TI"}
      }
    }
  }],
  "included": [
    {
      "type": "TI",
      "id": "TI3056",
      "attributes": {
        "serialNumber": "dS5udS9Ga1E1RQ",
        "manufacturer": "siemens",
        "value": 50.4,
        "unit": "celcius"
      }
    }
  ]
}
```

Listing 6: Code-Beispiel Request/Response mit JSON:API (Quelle: eigene Darstellung in Anlehnung an [51])

Eine weitere Möglichkeit die Response zu verändern, ist durch *Sparse Fieldsets* gegeben, die als Anfrage-Parameter übergeben werden können. Wie in Listing 7 zu sehen ist, kann durch die Angabe eines solchen Sparse Fieldsets in Form des Anfrage-Parameters *fields* eingeschränkt werden, welche Felder zurückgegeben werden sollen. [14, S. 15], [51]

```
GET /api/tcs?include=meter&fields[tc]=kp,meter&fields[TI]=value,unit HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "data": [{
    "type": "TC",
    "id": "TC3056",
    "attributes": {
      "kp": 1.877
    },
    "relationships": {
      "meter": {"data": {"id": "TI3056", "type": "TI"} }
    }
  }],
  "included": [
    {
      "type": "TI",
      "id": "TI3056",
      "attributes": {
        "value": 50.4,
        "unit": "celcius"
      }
    }
  ]
}
```

Listing 7: Code-Beispiel Sparse Fields JSON:API (Quelle: eigene Darstellung in Anlehnung an [51])

Die Vorteile von JSON:API im Gegensatz zu REST liegen bei der Reduzierung von Over- und Under-fetching durch Compound Documents und Sparse Fieldsets.

Durch die Angabe von Include-Parametern kann außerdem die Mehrfachnennung von Werten vermieden werden. Wenn beispielsweise aus einer Liste von Temperaturregelungen (TC) mehrere dieser Regelungen die gleiche Messung ansprechen sollen, tauchen die Attribute der Messung nur einmalig in der Response auf, während sie bei REST in jeder Darstellung einer Temperaturregelung angegeben wären.

Zudem bietet die JSON:API-Spezifikation auch die Möglichkeit für Paginierung und Sortierung sowie Filter, um die Anfragen weiter einzuschränken. Auch ein Error-Handling, welches zusätzlichen Kontext neben den HTTP-Fehlercodes liefert und die Angabe mehrere Fehlercodes auf einmal, ist möglich. [14, S. 15], [51]

Ein weiterer Vorteil ist wie bei REST die Möglichkeit des Caching von Ressourcen [52].

Ein Nachteil der JSON:API ist wie auch schon bei REST das Fehlen von Subscriptions. Weiterhin erweisen sich die schlechte Dokumentation und fehlende Ressourcen zur API im Internet als große Hürden bei der Implementierung. Es finden sich nur sehr wenige Webseiten, geschweige denn Bücher, die auf die API eingehen.

4.2.1.3 gRPC

gRPC wurde von Google entwickelt und nutzt die Eigenschaften von *RPC* (Remote Procedure Calls) und *Protobuf* (Protocol Buffers), in dem es diese kombiniert [14, S. 15]. Es wird von Größen wie Netflix, Cisco und dem Zahlungsdienstleister Square eingesetzt [48]. *Protobufs* sind ein serialisiertes, sprach- und plattformunabhängiges Dateiformat, welches auch durch Google entwickelt wurde und wie gRPC unter einer Open-Source-Lizenz läuft [53], [54].

Durch gRPC ist es einem Client - wie schon bei RPC - möglich, Methoden auf einen Server aufzurufen, wie als wenn diese lokal aufgerufen werden. Die Protobuf-Datei wird dabei als eine Art Schema genutzt, dass es ermöglicht, auf der Clientseite sogenannte *Stubs* zu erzeugen, welche die Methoden des Servers aufrufen können. Dadurch ist es egal unter welcher Programmiersprache der Client läuft[6], da die Stubs für die Umsetzung der Funktionen in das Protobuf-Format zuständig sind, welches dann binär übertragen wird. [14, S. 15–16], [55]

Da sich gRPC stark von den anderen hier vorgestellten Technologien unterscheidet, wird an dieser Stelle eine Protobuf-Datei für das Lesen eines Reglers wie aus dem Beispiel in Listing 5 verwendet. Der Aufbau einer solchen Protobuf-Datei wird im Listing 8 dargestellt.

```
1  syntax = "proto3";
2
3  package controllerPackage;
4
5  service controller {
6         rpc readController (google.protobuf.Empty) returns (controllerItem);
7  }
8
9  message controllerItem {
10        string id = 1;
11        double kp = 2;
12        double tn = 3;
13        double tv = 4;
14        string controllerId = 5;
15        string meterId = 6;
16 }
```

Listing 8: Beispielhafte Protobuf-Datei (Quelle: eigene Darstellung)

Die Vorteile von gRPC sind vor allem durch die binäre Übertragung und die Pflicht zu HTTP/2 gegeben. Die Daten sind durch die binäre Übertragung kleiner, als dies zum Beispiel bei JSON der Fall ist und HTTP/2 bietet eine wesentlich bessere Komprimierung sowie Multiplexing, was

[6] Genau genommen unterstützt gRPC nur die Erzeugung von Stubs für C#, C++, Dart, Go, Java, Kotlin, Node, Objective-C, PHP, Python und Ruby.

das Senden über mehrere Datenströme auf einmal erlaubt [56], [57]. Von diesen Vorteilen können zwar auch die anderen Technologien bei entsprechender Implementierung profitieren, jedoch ist dies bei gRPC immer der Fall. Ein weiterer Vorteil liegt in der Unterstützung von bidirektionalen Verbindungen, durch HTTP/2. [14, S. 17], [58]

Hiermit ist es möglich, Subscriptions ohne den Einsatz weiterer Tools zu realisieren.

Als Vorteil kann auch die Betreuung der *Library* (Programmbibliothek) durch ein großes Unternehmen wie Google angesehen werden. Es gibt zudem auch nur eine Library für alle unterstützten Sprachen und man muss nicht auf Libraries setzen, die beispielsweise durch kleine Teams betreut werden. Dies sorgt für eine schnelle Behebung von Fehlern und Sicherheitslücken.

Zu den Nachteilen gehört, dass man sich erst in die Nutzung eines Schemas einarbeiten muss, dessen Technologie noch recht neu ist. Auch ein Error-Handling durch HTTP, wie man es von REST kennt, ist nicht vorhanden, sodass dies eigenständig implementiert werden muss. [59]

Durch die binäre Übertragung und Protobufs ist außerdem auch kein Support durch Browser gegeben und wird auch in Zukunft wohl nicht folgen, da das Einsatzgebiet eher die Kommunikation zwischen Microservices darstellt. [14, S. 17]

Durchaus kann man die Pflicht zu HTTP/2 auch als Nachteil sehen. Zwar wurde HTTP/2 bereits im Jahr 2015 eingeführt, jedoch ist der Support noch nicht durch alle Server gegeben, wie Abbildung 7 zeigt.

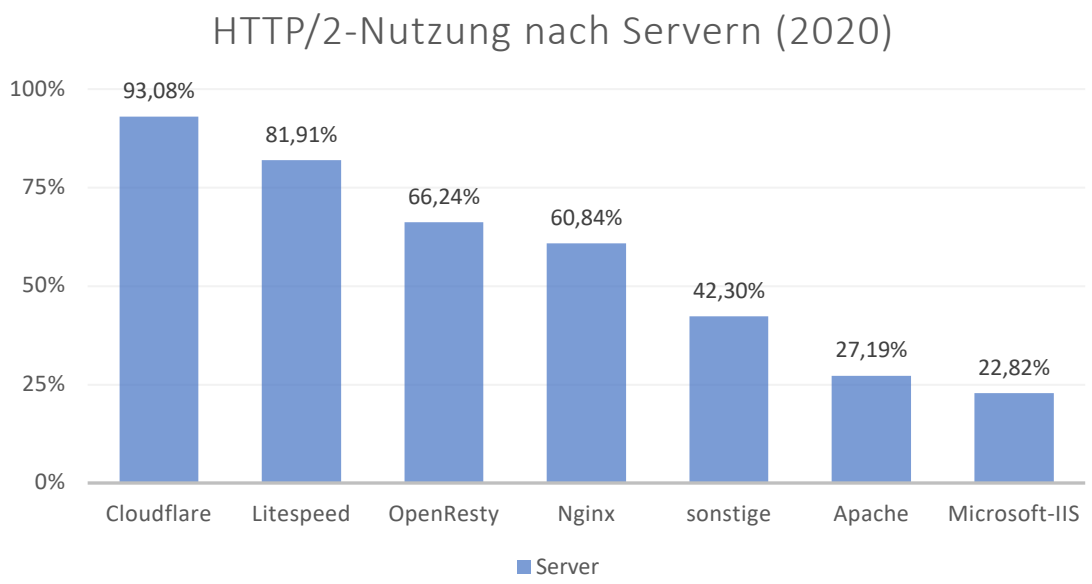


Abbildung 7: Nutzung von HTTP/2 nach Servern (Quelle: eigene Darstellung in Anlehnung an [60])

4.2.1.4 GraphQL

Bei GraphQL handelt es sich um eine plattformunabhängige *Query Language* (Abfragesprache), durch die es möglich ist, über einen einzigen Endpunkt verschiedene Abfragen zu tätigen. Dies zeigt den Unterschied zu REST, wo für jede Ressource eine eindeutige URI zugewiesen ist. Bei GraphQL wird auf ein vorher definiertes Schema gesetzt, nach denen sich die Queries richten.

In Listing 9 wird ein Beispiel für ein solches Schema gezeigt. [14, S. 18], [61]

```
type Query {
  TCs: [TC]
}

type TC {
  id: String!
  kp: Float!
  tn: Float!
  tv: Float!
  controller: Valve!
  meter: TI!
}

type Valve {
  id: String!
  value: Float!
  manufacturer: String!
  serialNumber: String!
}

type TI {
  id: String!
  manufacturer: String!
  serialNumber: String!
  value: Float!
  unit: String!
}
```

Listing 9: Beispiel für ein Schema unter GraphQL (Quelle: eigene Darstellung)

Durch die Verweise innerhalb von Objekttypen auf andere Objekttypen werden Beziehungen abgebildet, ähnlich wie es bei JSON:API über die Relationships möglich ist. Das Schema stellt dabei eine Sammlung aller möglichen Abfragen dar. [14, S. 18], [61]

Die eigentliche Query wird von dem Client über den Body einer HTTP-POST-Anfrage an den Server gesendet. Wie in Listing 10 zu sehen ist, ermöglicht es dies dem Client genau zu bestimmen, welche Daten er abfragen will.

```
POST /graphql HTTP/1.1

{
  TCs (id="TC3056"){
    id
    kp
    meter {
      id
      value
      unit
    }
  }
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "data": {
   "TCs": {
     "id": "TC3056",
     "kp": 1.877,
     "meter": {
       id: "TI3056",
       value: 50.4,
       unit: "celcius"
     }
   }
}
```

Listing 10: Beispiel für eine Request/Response bei GraphQL (Quelle: eigene Darstellung)

Diese Möglichkeit die Abfragen dynamisch einzuschränken, vermindert wie schon bei JSON:API das Over- und Under-fetching. Weiterhin bietet GraphQL die Möglichkeit zur Verwendung von Variablen, was vordefinierte Queries ermöglich, die zusammen mit Variablenwerten versendet werden. Diese Variablen werden dann durch die Variablenwerte ersetzt, was eine manuelle Änderung der Queries unnötig macht und somit eine einfache Möglichkeit zur Anpassung liefert. Um die Menge an Daten beim Versenden von Abfragen zu minimieren, bietet GraphQL unter anderem Aliasse, welche dazu dienen, Konflikte bzw. Verwechslungen bei mehrfach angefragten Teilen von Daten zu vermeiden. Durch Fragmente ist es außerdem möglich Sub-Queries zu erstellen, welche in der Haupt-Query mehrfach wiederverwendet werden können.[14, S. 19, 44–49]

Da GraphQL das HTTP-Verb POST für alle Anfragen an den Server verwendet, findet eine andere Nutzung der Verben statt als bei REST. Bei REST wird die Abfrage von Daten über die GET-Methode gelöst. Während in REST die POST-, PUT- und DELETE-Methode zur Erzeugung, Veränderung und zur Löschung von Daten genutzt werden, wird dies in GraphQL über *Mutations* gelöst. Bei Mutations wird wie bei Queries eine Struktur der Daten gesendet welche Variablen nutzt. Die Werte der Variablen werden zusammen mit der Mutation gesendet und durch den Server verarbeitet.[14, S. 54–55]

Eine weitere Funktion, die GraphQL im Gegensatz zu REST und JSON:API unterstützt, sind Subscriptions. GraphQL kümmert sich dabei nicht um die konkrete Umsetzung, sondern stellt lediglich Möglichkeiten zur Anmeldung einer Subscription zur Verfügung. Mittels GraphQL wird dem Server mitgeteilt, für welche Daten der Client Updates erhalten will, während die Implementierung und die Auswahl der dafür genutzten Technologie offen sind. Eine mögliche Technologie wäre hier zum Beispiel *WebSockets*, wobei jede Push-Technologie grundsätzlich möglich ist, was sich bei Facebook zeigt, welches *MQTT* (Message Queuing Telemetry Transport) nutzt. Diese voreingebaute Unterstützung von Subscriptions ist ein Vorteil gegenüber REST oder JSON:API. [14, S. 55–57], [62]

Auch im Bereich Dokumentation bietet GraphQL einige Vorteile. Ein Vertreter für dynamische Dokumentation kommt von Facebook selbst und nennt sich GraphiQL. Dieses Tool bietet eine einfache Möglichkeit der Dokumentation und bedarf nur wenig Einarbeitungszeit. Zusätzlich ermöglicht es auch das Testen des Codes, Textvervollständigung und Debugging. [14, S. 98]

Die Funktionsweise von GraphQL ermöglicht bis zu einem gewissen Maße die getrennte Entwicklung von *Frontend* (Präsentationsebene) und *Backend* (z.B. Datenzugriffsebene). So können im Backend weitere Daten zur Verfügung gestellt werden, die vom Frontend bis zur Implementierung ignoriert werden können durch unveränderte Abfragen.

Wie bereits erwähnt, bezieht man in GraphQL nur die Daten, die wirklich von Interesse sind. Diese Funktion bietet auch JSON:API, jedoch schafft GraphQL dies mit einer niedrigeren Menge an Daten als dies bei JSON:API der Fall ist, wie man durch den Vergleich der Code-Beispiele aus Listing 7 und Listing 10 sehen kann.

Einer der Nachteile von GraphQL entsteht dadurch, dass es nur einen Endpunkt gibt. In vielen APIs lässt sich der Zugang zu Ressourcen für ganze Pfade oder einzelne Endpunkte sehr genau regeln. Um dies in GraphQL zu lösen, bietet sich nur die Auslagerung der Autorisierung an *Middleware* (Zwischenanwendung) an, welche diese Aufgabe übernimmt. [14, S. 93–95]

Als Nachteil kann - wie auch bei gRPC - die Einarbeitungszeit gesehen werden. Durch die Schemas ergibt sich ein erhöhter Aufwand bei der Einarbeitung, wohingegen REST eine sehr einfache Umsetzung bietet.

Error-Codes werden bei GraphQL nicht durch die HTTP-Statuscodes abgebildet. Dadurch das GraphQL auf die HTTP-Methode POST setzt, wird bei Fehlern trotzdem der HTTP-Statuscode *200 OK* zurückgegeben, da die Umsetzung der HTTP-Methode erfolgreich war und die Daten, also in dem Fall die Query bzw. Mutation erfolgreich gesendet wurde. Dementsprechend ist es erforderlich ein eigenes Error Handling zu implementieren.

4.2.1.5 Verbreitung und Nutzung

Im Folgenden wird ein kurzer Überblick über die Verbreitung der Technologien gegeben. Einen ersten Eindruck bezüglich der Verbreitung bietet Google Trends. Google Trends gibt eine Übersicht darüber, wie oft bestimmte Suchbegriffe in einem festgelegten Zeitraum gesucht wurden. Somit stellt Google Trends zwar nicht direkt die Verbreitung bzw. Nutzung der APIs dar, zeigt jedoch das Interesse an den Technologien, woraus sich in vielen Fällen Prognosen erstellen lassen. [63]

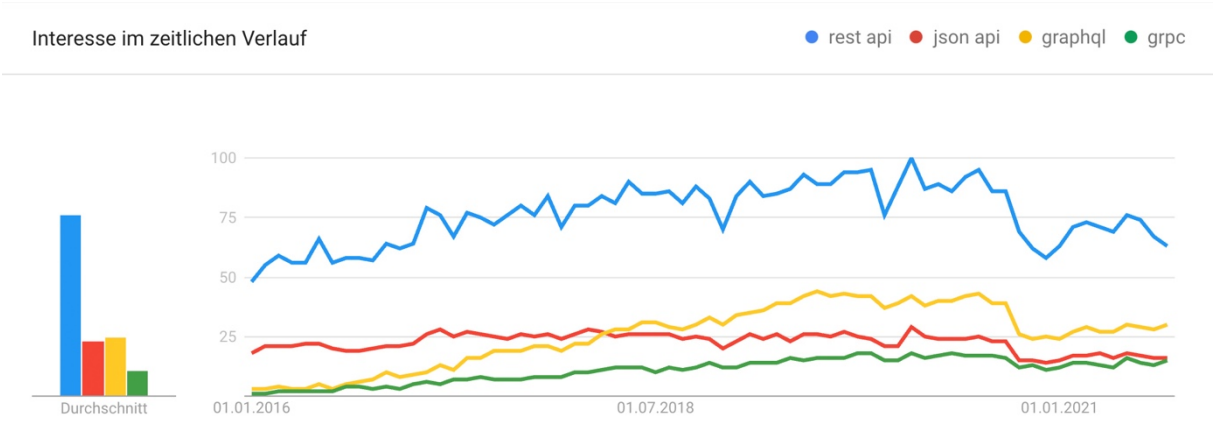


Abbildung 8: Interesse an den Web-APIs nach Google Trends (Quelle: eigene Darstellung in Anlehung an [64])

Im Google-Trends-Chart aus Abbildung 8 wurde der Startpunkt für die Betrachtung auf den 01.01.2016 festgelegt, da spätestens ab diesen Zeitpunkt alle vier Technologien öffentlich zugänglich waren. Das Ergebnis zum Thema *json api* sollte jedoch zwiespältig betrachtet werden, da sich der Suchbegriff durch die unglückliche Namensgebung der JSON:API nicht ausschließlich auf die API-Spezifikation beziehen muss, sondern beispielsweise auch auf APIs, die das JSON-Format nutzen bzw. ausgeben.

Einen anderen Blick bietet der *State of API Report*, welcher auszugsweise in Abbildung 9 dargestellt ist. Die Umfrage stammt von der Firma *SmartBear*, welche Projekte wie Swagger und OpenAPI betreut [65]. Bei dieser Umfrage wurden über 1500 API-Nutzer und Kunden aus verschiedensten Teilen der Industrie befragt, sodass diese Umfrage ein besseres Bild über das Produktionsumfeld gibt, als dies bei Google Trends der Fall ist. [66].

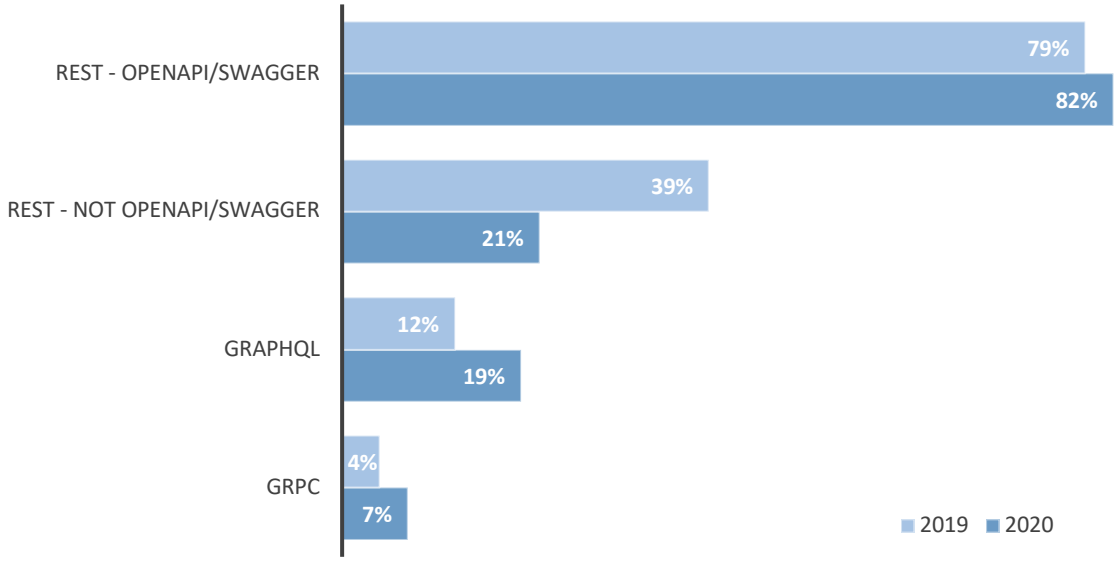


Abbildung 9: Bevorzugte API-Designs laut SmartBear (Quelle: eigene Darstellung in Anlehnung an [66], [67])

Im Vergleich zur Umfrage aus dem Vorjahr zeigt sich, dass immer mehr der Befragten auf moderne Technologien wie gRPC und GraphQL setzen, während die Nutzung von REST ohne Implementierung über Swagger/OpenAPI stark rückläufig ist, aber immer noch mehr zum Einsatz kommt als GraphQL oder gRPC. Die Verwendung von REST in Kombination mit Swagger/OpenAPI hat im Vergleich zum Vorjahr sogar etwas zugenommen.

Abbildung 10 zeigt eine weitere Entwicklerumfrage von *RapidAPI*, einem der größten Marktplätze für APIs. Die Umfrage gibt einen Überblick über das aktuelle Nutzungsverhalten der Befragten, aufgeschlüsselt nach den APIs. An der Umfrage nahmen über 2000 Entwickler teil, von denen 50 % professionelle Entwickler sind. Das Ergebnis zeichnet dabei ein ähnliches Bild ab, wie es aus der Umfrage aus Abbildung 9 zu entnehmen ist.

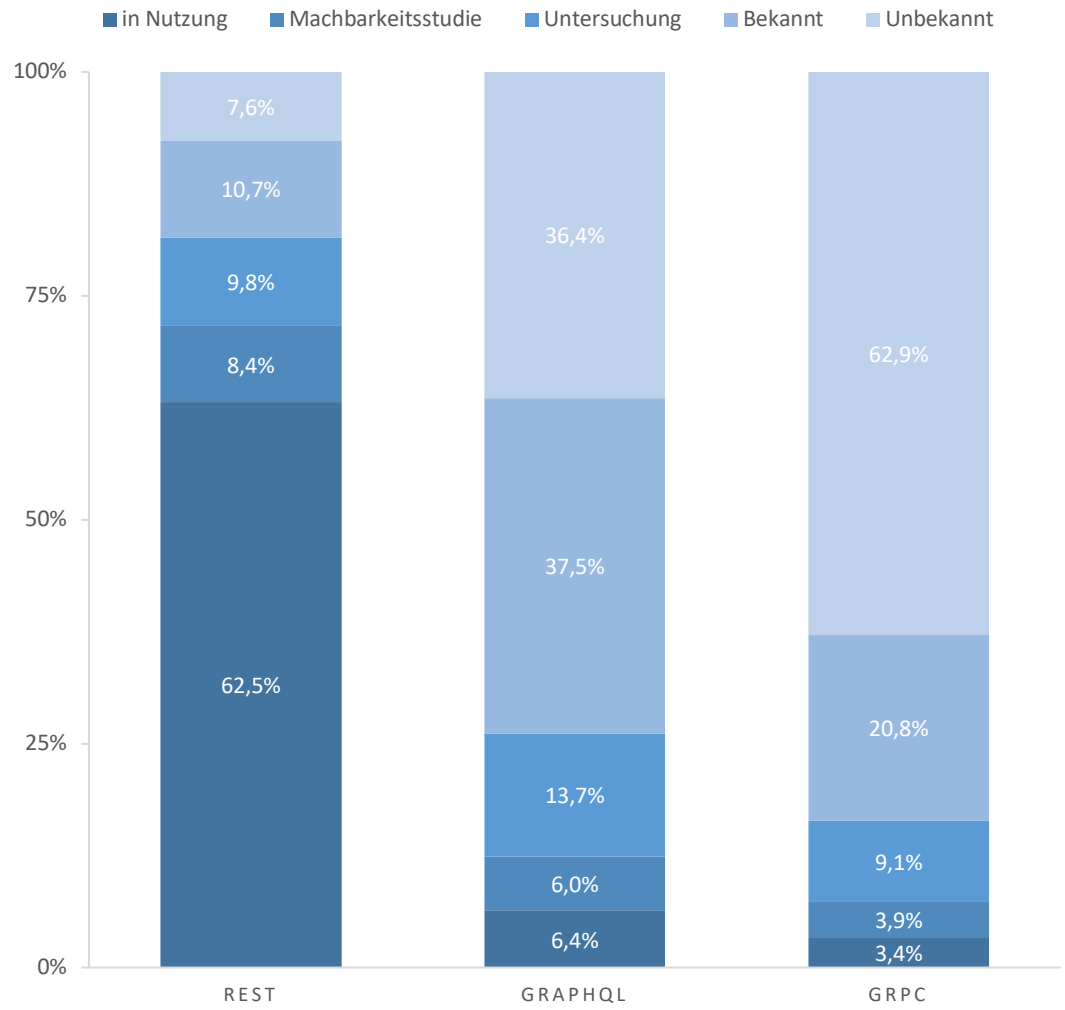


Abbildung 10: Nutzung von API-Designs laut RapidAPI (Quelle: eigene Darstellung in Anlehnung an [68, S. 5])

Als letzte Umfrage zu diesem Thema dient der *State of the API Report* von der Firma *Postman*. Postman ist der Hersteller des gleichnamigen Programms, welches zum Entwickeln und Testen von APIs gedacht ist und laut Aussagen von Postman durch über 15 Millionen Entwicklern genutzt wird [69]. An der Umfrage nahmen 13586 Angehörige der API-Branche teil, welche unter anderem gefragt wurden, mit welchen Technologien sie vertraut sind. In Abbildung 11 lässt sich eine Zusammenfassung der Ergebnisse entnehmen. Die Studie ergab außerdem, dass Teilnehmer mit mehr als 6 Jahren Berufserfahrung eher REST nutzen, als dies Teilnehmer mit weniger Berufserfahrung tun.

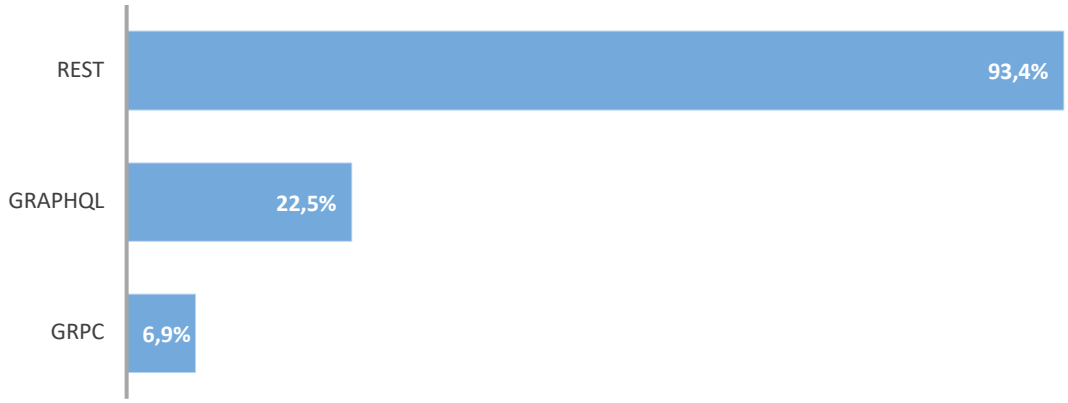


Abbildung 11: Vertrautheit mit API-Technologien (Quelle: eigene Darstellung in Anlehnung an [70])

Bei allen vier Statistiken zeichnet sich ein klares Bild ab. Technologien wie GraphQL und gRPC sind von immer größerem Interesse bei den Entwicklern. REST verliert zwar Anteile, ist aber von den vier vorgestellten API-Designs nach wie vor das Bekannteste. Die Verbreitung der JSON:API lässt sich nur schwer einschätzen, jedoch ist es ein klares Zeichen, dass sie in den drei Entwicklerumfragen nicht als eine mögliche Option gelistet wurde. Auch bei Google trifft man bei der Suche nach der JSON:API sehr oft auf Ergebnisse, die nichts mit der API-Spezifikation zu tun haben, was man als weiteres Zeichen für die geringe Verbreitung der API sehen kann.

4.2.2 Festlegung

Im Gegensatz zu den anderen Technologien eignet sich gRPC am wenigsten für den vorgesehenen Einsatzzweck. gRPC setzt als einzige der Technologien auf eine servicebasierte Kommunikation, wohingegen die anderen Technologien auf ressourcenbasierte Kommunikation setzen. Da OPC UA durch den Einsatz von Services ähnlich arbeitet, wäre gRPC von diesem Standpunkt aus betrachtet sogar die am besten geeignete Technologie. Der fehlende Browser-Support und die noch recht unbekannte Technologie sprechen aber gegen den Einsatz dieser Technik, da damit die Bedingungen der Anforderungen NFA_07 und auch teilweise die von NFA_04 und NFA_05 verletzt werden.

Gegen den Einsatz der JSON:API sprechen die bereits vorher angesprochene geringe Verbreitung und die unzureichende Dokumentation. Auch hier werden die Bedingungen, der Anforderungen NFA_04 und NFA_05 nicht erfüllt.

Die beiden Technologien, die übrig bleiben, sind GraphQL und REST. GraphQL ist zwar der am stärksten etablierte von denen neuen Technologieansätzen, jedoch ist REST nach wie vor die Bekannteste und am meisten genutzte Technologie, wie sich aus 4.2.1.5 entnehmen lässt.

Durch die vorher zu definierenden Schemas eignet sich GraphQL eher schlecht für dynamische Informationsmodelle, wie sie bei OPC UA vorzufinden sind. Hier ist REST offener und bietet auch allgemein einen verständlicheren Ansatz, wenn es darum geht, sich mit der Technologie vertraut zu machen. REST ist die einfachere der beiden Technologien, was sich unter anderem auch durch Verbreitung der zeigt. Die nichtfunktionale Anforderung NFA_05 verlangt einen verständlichen, also auch einfachen Code. In den Umfragen aus 4.2.1.5 wurde auch festgestellt, wie wichtig eine einfache Nutzung der Technologien ist. So wurde im *State of API Report* von SmartBear festgestellt, dass eine einfache Handhabung der API an erster Stelle steht, wenn es um die wichtigsten Eigenschaften einer API geht [67]. Auch nach dem *State of the API Report* von Postman ist eine zu hohe Komplexität eines der größten Hindernisse beim Verwenden von APIs [70].

Für GraphQL spricht wiederum die bessere Performance gegenüber REST, was sich auch durch die Masterarbeit *Untersuchung der Performance einer Web-API mit GraphQL und REST*[7] von Ferdinand Malcher aufzeigen lässt. Dabei wurde die Performance einer Web-API mit REST und GraphQL verglichen. GraphQL arbeitet in den untersuchten Szenarien je nach Menge der HTTP-Anfragen schneller oder mindestens genauso schnell wie REST. [71, S. 101]

Da diese Arbeit lediglich einen möglichen Ansatz der Implementierung und kein fertiges Produkt liefert, kann und sollte dieser Ansatz durch Dritte fortgeführt werden. Um dies zu gewährleisten, braucht es eine einfache und verständliche Implementierung, deren Komplexität nicht noch durch die diffizilen Schemas und die schlechte Eignung von GraphQL für dynamische Informationsmodelle verstärkt wird. Insgesamt eignet sich REST damit besser für den vorgesehenen Einsatzzweck.

4.3 Auswahl Push-Technologien

Die Server-Push-Technologie dient in diesem Projekt zum Abbilden der Subscriptions aus OPC UA. Im Folgenden werden verschiedenste Server-Push-Technologien verglichen und anschließend wird eine passende Technologie für das Projekt ausgewählt.

4.3.1 Vergleich

In diesem Abschnitt findet eine Vorstellung der Server-Push-Technologien statt, mit Bezug auf ihre Stärken und Schwächen sowie einen Überblick über die Verbreitung der Techniken.

[7] [71]

4.3.1.1 HTTP Polling

Beim HTTP Polling sendet der Client periodisch Anfragen an den Server, ob Updates für die Ressource bereitstehen. Die Zeit zwischen zwei Anfragen ist der Polling-Intervall. Sollte bei einer Anfrage ein Update der Ressource festgestellt werden, wird diese von dem Client übernommen. Sollte die Ressource unverändert sein, dann steht keine aktuellere Ressource zur Verfügung und der Polling Intervall beginnt von neuem. [72]

Die Darstellung einer solchen Abfolge ist in Abbildung 12 als Sequenzdiagramm wiedergegeben.

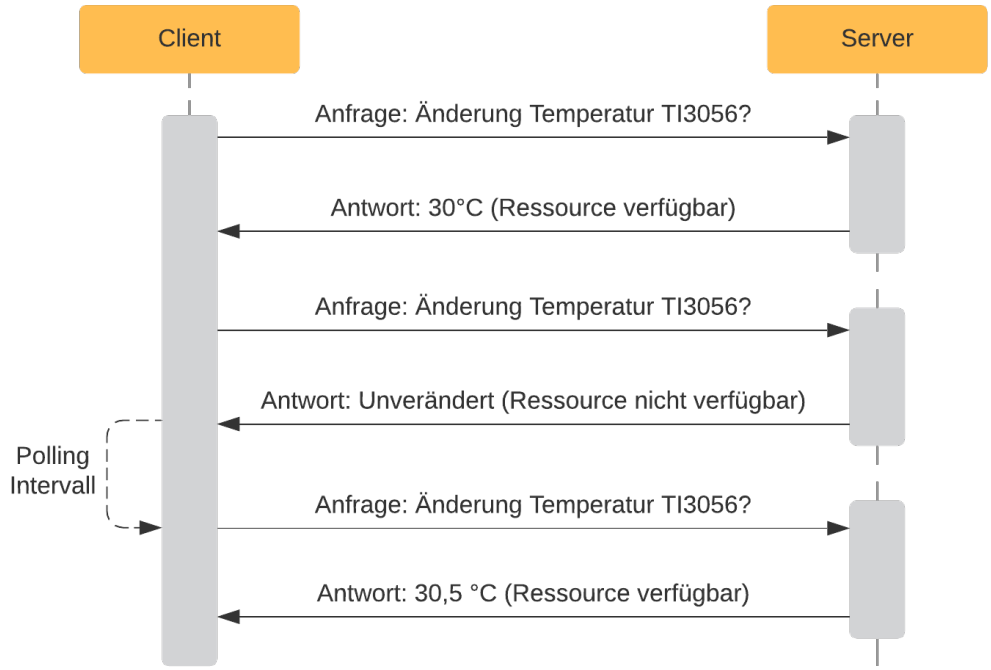


Abbildung 12: Ablauf von HTTP Polling (Quelle: eigene Darstellung)

Der Vorteil dieser Technologie ist, dass sie sich sehr einfach implementieren lässt, da es sich nur um einen *Loop* (Schleife) aus drei Schritten handelt:

1. Der Client sendet eine HTTP-Anfrage.
2. Der Client erhält die neue Ressource und verarbeitet sie oder er erhält sie nicht.
3. Warten für eine festgelegte Zeit (Polling-Intervall).

Der große Nachteil dieser Technologie sind die unnötigen Anfragen, da immer eine Anfrage und Antwort gesendet wird, auch wenn sich die Ressource nicht geändert hat.

4.3.1.2 HTTP Long Polling

Bei HTTP Long Polling sendet der Client eine Anfrage an den Server und die daraus resultierende geöffnete Verbindung bleibt bestehen, bis der Server eine neue Ressource bereitstellen kann. Sollte der Server die Ressource nicht bereitstellen können, bleibt die Verbindung so lange bestehen, bis das *Connection Timeout* (Zeitüberschreitung der Verbindung) erreicht wird. Nach Ablauf des Polling-Intervalls sendet der Client eine erneute Anfrage für die Ressource. Dieser Ablauf wird in Abbildung 13 dargestellt. [73]

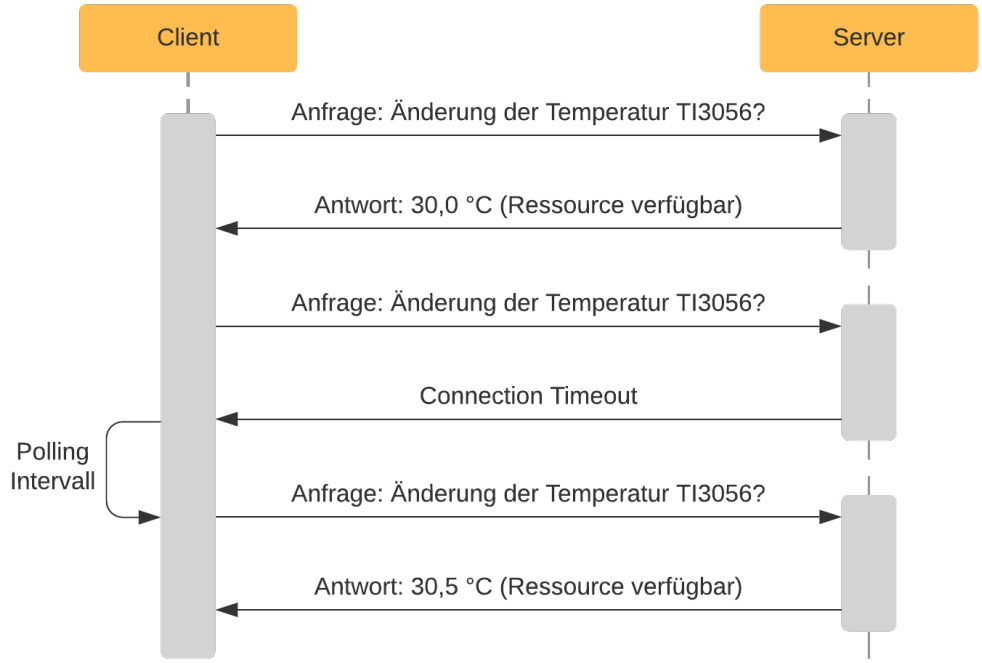


Abbildung 13: Ablauf von HTTP Long Polling (Quelle: eigene Darstellung)

Im Vergleich zu HTTP Polling entstehen so weniger Anfragen vom Client an den Server, jedoch muss der Client immer noch den Status regelmäßig und selbstständig abfragen. Außerdem wird durch die bestehende Verbindung dauerhaft eine TCP-Verbindung für jede einzelne abzufragende Ressource offengehalten, womit der Server weiter belastet wird und bei zu vielen Anfragen schnell an die Grenze seiner maximal möglichen TCP-Verbindungen kommen könnte.

4.3.1.3 WebSockets

WebSocket ist ein Protokoll, welches im Jahr 2011 standardisiert wurde. Es ist vollduplexfähig, was bedeutet, dass Daten in beide Richtungen zur selben Zeit übertragen werden können [74]. Die Kommunikation findet dabei über TCP statt, wird jedoch über einen HTTP-Handshake eröffnet. Dafür sendet der Client eine HTTP-Anfrage für die Öffnung einer WebSocket-Verbindung, welche der Server dann mit dem HTTP-Status *101 Switching Protocols* bestätigt. Anschließend wird eine TCP-Verbindung aufgebaut, über die der Client und der Server kommunizieren können. In Abbildung 14 wird der Verbindungsaufbau und eine beispielhafte Kommunikation aufgezeigt. In diesem Beispiel wird zwar nur durch den Client eine Anfrage für die Ressource gesendet, jedoch ist es durch die bestehende vollduplexfähige Verbindung möglich, dass der Server den Client auch ohne vorherige Anfrage Daten sendet. [75]

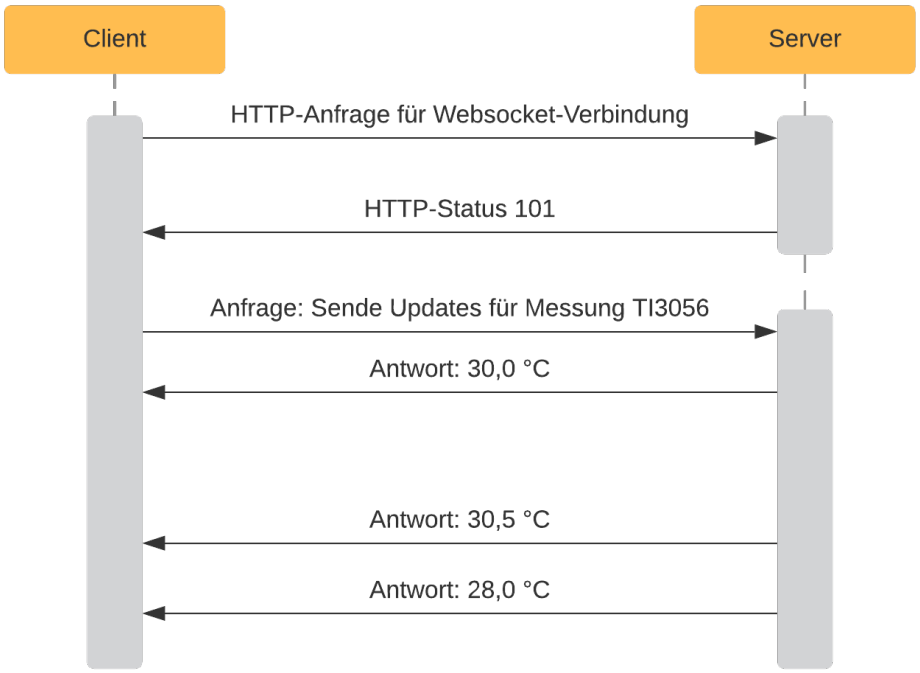


Abbildung 14: Beispielhafte Kommunikation WebSockets (Quelle: eigene Darstellung)

Der Vorteil von WebSockets besteht darin, dass der Client nicht ständig Anfragen an den Server senden muss, sondern die Ressource erhält, sobald sie verfügbar ist. So ist es möglich eine einzige Anfrage für Updates zu einer Ressource zu stellen und dauerhaft Daten bei einer Änderung der Ressource zu erhalten. Dadurch können die Anfragen im Vergleich zu HTTP Long Polling weiter reduziert werden.

Der Nachteil ist die dauerhaft aufrechterhaltene Verbindung. Weiterhin bietet das Protokoll keine vordefinierte Möglichkeit eines erneuten automatischen Verbindungsaufbaus bei einem unerwarteten Abbruch der Verbindung [76].

4.3.1.4 Server-Sent Events (SSE)

Durch Server-Sent Events (SSE) ist es dem Client möglich, Updates von einem Server über eine HTTP-Verbindung in Form von Push-Nachrichten zu erhalten. Im Gegensatz zu der bidirektionalen Kommunikation von WebSockets, ist SSE auf einseitige Kommunikation ausgerichtet und nutzt für die Übertragung HTTP. Der Client nutzt ein *EventSource-Objekt,* um Daten vom Server zu empfangen. Dem Server wird über eine HTTP-Anfrage mitgeteilt, dass der Client Updates für die Ressource zu einer bestimmten URL gesendet bekommen will. Anschließend sendet der Server die Updates an diese URL, sobald diese auftreten. Dieser Ablauf ist in Abbildung 15 dargestellt. [77]–[79]

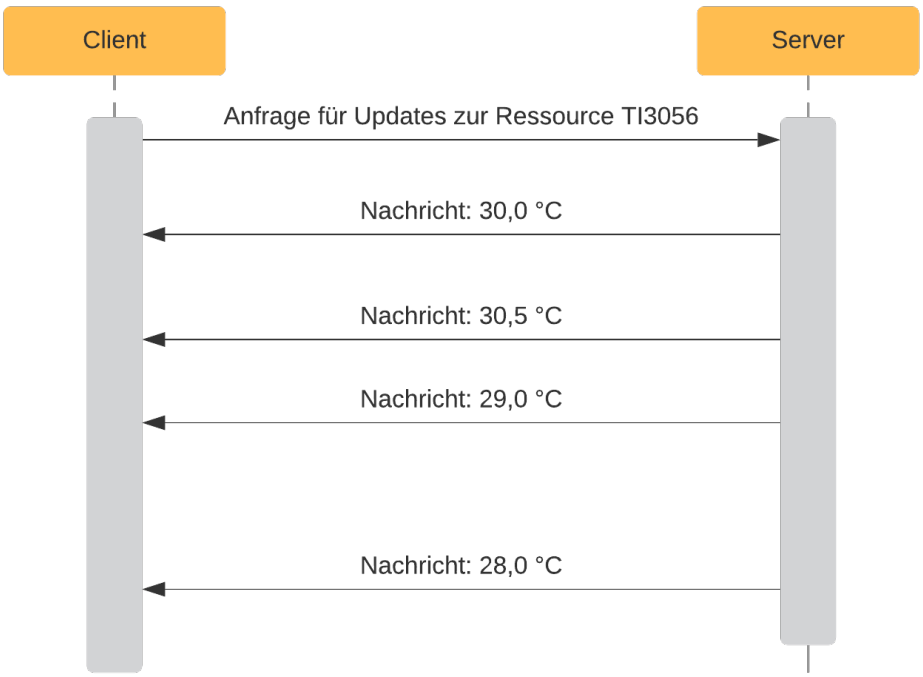


Abbildung 15: Ablauf Server-Sent Events (Quelle: eigene Darstellung)

Server-Sent Events eignen sich vor allem dann, wenn Updates von sich regelmäßigen änderten Ressourcen erhalten werden sollen, sonst aber keine weitere Kommunikation vom Client zum Server notwendig ist. Ein Vorteil von SSE gegenüber WebSockets besteht in der besseren Skalierbarkeit. Wenn eine Ressource von mehreren Clients angefragt wird, kann der Server diese ähnlich einem *Broadcast* (Rundruf) an alle Clients auf einmal senden, während bei WebSockets jeder Client über seine exklusive WebSocket-Verbindung bedient wird. Je nach Einsatzszenario kann die einseitige Kommunikation auch als Nachteil gesehen werden.

Des Weiteren ist SSE von der maximalen Anzahl von offenen Verbindungen zu einer Domain betroffen, welche bei aktuellen Browsern bei maximal sechs Verbindungen liegt. Dieser Nachteil kann aber durch die Nutzung von HTTP/2 umgangen werden. [79]

4.3.1.5 Push API

Bei der Push API meldet sich der Client für Updates zu einer Ressource an. Der Server sendet diese Updates allerdings nicht direkt, sondern sie werden an einen Push Service gesendet, der sich um die Verteilung der Nachrichten kümmert. [80], [81]

In Abbildung 16 wird dieser Aufbau schematisch dargestellt. Wie anhand der Skizze zu sehen ist, ist es möglich, Nachrichten von mehreren Servern an einen Client oder die Nachricht eines einzelnen Servers an mehrere Clients zu senden. Dies stellt einen der Vorteile von der Push API dar, da somit die Menge an Verbindungen, welche vom Client oder Server ausgehen, reduziert werden können. [82]

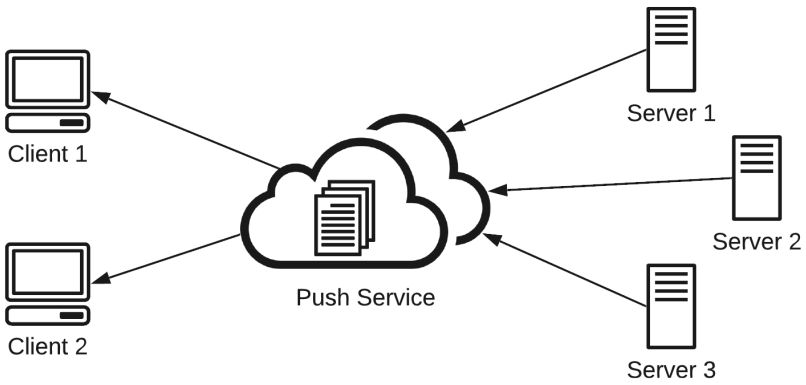


Abbildung 16: Schematische Darstellung der Kommunikation bei der Push API (Quelle: eigene Darstellung)

Ein weiterer Vorteil ist, dass Nachrichten vom Server gesendet werden können, auch wenn der Client gerade nicht erreichbar ist, da die Daten durch den Push Service zum Client gesendet werden, sobald dieser wieder erreichbar ist.

Einen Nachteil stellt der Push Service selbst dar, der ein fremder Server ist, weswegen die Nachrichten verschlüsselt gesendet werden sollten. Weiterhin sind bei Problemen mit dem Push Service alle Verbindungen von diesem Problem betroffen, weswegen die Kommunikation aller Geräte gleichermaßen eingeschränkt wäre.

4.3.1.6 Verbreitung und Nutzung

Für einen Überblick über die Technologien wird ein Google-Trends-Chart genutzt, welches einen Aufschluss über das Suchverhalten gibt und damit das Interesse der Nutzer widerspiegelt.

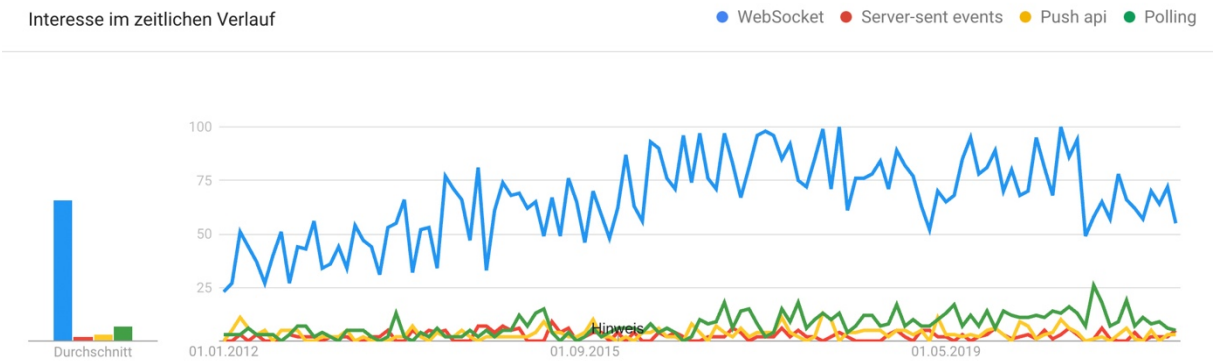


Abbildung 17: Google Trends Chart zu Push-Technologien (Quelle: eigene Darstellung in Anlehnung an [83])

Einen Blick aus Entwicklersicht bietet der *State of the API Report* von Postman, auf den bereits in 4.2.1.5 Bezug genommen wurde. Dabei wurden die Teilnehmer befragt, mit welchen Techniken sie vertraut sind. Betrachtet man die hier vorgestellten Technologien dann zeigt sich, dass wesentlich mehr Entwickler mit WebSockets (22,4 %) umgehen können, als dies bei Server-Sent Events (6,2 %) der Fall ist. Die Polling-Technologien sowie die Push API wurden bei der Umfrage nicht aufgeführt. [70]

Einen besseren Überblick der Nutzung im World Wide Web bietet die Arbeit *WebSocket Adoption and the Landscape of the Real-Time Web*[8] vom April 2021. In dieser Arbeit wurden die *Tranco Top 1 Million* genutzt, welche eine Liste der am meisten besuchten Webseiten innerhalb der letzten 30 Tage darstellen, die laut Aussage der Herausgeber sicherer gegen Manipulationen ist als dies bei anderen Listen der Fall ist [85]. Über diese Webseiten wurde ein *Crawler* geschickt, was ein Programm ist, dass das Internet systematisch nach Daten durchsucht und diese analysiert. Für die Prüfung auf Polling und Long Polling wurde auf Grund des Aufwands eine Testmenge von 4000 Webseiten gebildet. Von diesen Webseiten nutzten 14,8 % Polling-Technologien. Bei der Unterscheidung zwischen den Polling-Technologien wurde festgestellt, dass lediglich eine dieser 4000 Webseiten auf Long Polling setzte.

[8] [84]

Für Server-Sent Events und WebSockets konnte die gesamte Menge der Webseiten geprüft werden. Insgesamt nutzten davon 6,3 % WebSockets, während nur 0,05 % Server-Sent Events einsetzten. Im Jahr 2018 lag die Verwendung von WebSockets nur bei 1,6 bis 2,5 %, was bedeutet, dass die Nutzung signifikant zugenommen hat. [84]

Diese Ergebnisse deuten an, dass Polling trotz dessen Einschränkungen weiterhin eine wichtige Rolle spielt. Eine Betrachtung der moderneren Technologien zeigt, dass WebSockets eine immer größer werdende Rolle spielen, während Server-Sent Events eine eher untergeordnete Rolle dabei einnehmen.

4.3.2 Entscheidung

HTTP Polling und Long Polling kommen auf Grund der eingeschränkten Technologie und dem *Overhead*, also Daten, welche nicht zu den Nutzdaten gehören, für dieses Projekt nicht in Frage. Die Push API bietet zwar einen sehr modernen Ansatz und einige Vorteile, findet jedoch eher Anwendung bei Benachrichtigungen in Browsern und Smartphones und nicht bei fortlaufend in kurzen Intervallen gesendeten Daten. Server-Sent Events eignet sich hingegen besser für diesen Einsatz. Gegen die Technologie spricht die aktuell geringe Nutzung, während WebSockets sehr gut etabliert sind und deren Verwendung weiter zunimmt. Damit eignen sich WebSockets besser, um die nichtfunktionale Anforderung NFA_04 zu erfüllen. Weiterhin bietet der bidirektionale Datenaustausch bei WebSockets eine Möglichkeit, die Anforderung FA_16 zu erfüllen, was bei SSE nicht gegeben ist. Aus diesen Gründen ist WebSockets für dieses Projekt die geeignetere Push-Technologie.

4.4 Auswahl des Authentifizierungsverfahrens

Bei der Authentifizierung wird das *HTTP Basic Authentication* genutzt, da dies ein einfach zu implementierendes Verfahren ist, dass durch viele Bibliotheken und Werkzeuge unterstützt wird. Bei HTTP Basic Authentication wird der Nutzername und das Passwort getrennt durch einen Doppelpunkt mit dem Kodierungsverfahren *Base64* kodiert und anschließend im HTTP-Header unter dem Anfrage-Parameter *Authorization* übertragen. [17, S. 142]

Da sich eine Base64 Kodierung ohne Probleme umkehren lässt, werden der Nutzername und das Passwort im Grunde für jeden lesbar übertragen. Um Verbindungen abzusichern bietet es sich an, HTTPS in Verbindung mit HTTP Basic Authentication zu nutzen. Da der Fokus dieser Arbeit nicht auf dem Thema kryptographische Absicherung liegt und sich eine Verschlüsselung über TLS, also das für HTTPS eingesetzte Protokoll, später problemlos nachrüsten lässt, wird auf dessen Implementierung vorerst verzichtet.

4.5 Repräsentationsformat

JSON ist als Format bei öffentlichen Web-API weiter verbreitet als XML [17, S. 97]. Ein weiterer Indikator dafür sind die Strack-Overflow-Trends, laut deren zum Thema JSON bei weitem mehr Fragen gestellt werden, als dies bei XML oder anderen Repräsentationsformaten der Fall ist [86]. Somit fällt die Wahl des Datenformats auf JSON.

Um JSON besser für die Unterstützung von Hypermedia zu gestalten, wurden verschiedene auf JSON basierende Hypermedia-Formate entwickelt. Die Wahl für dieses Projekt traf dabei auf *HAL* (Hypermedia Application Language), das vor allem durch seine einfache Umsetzung besticht [17, S. 92]. Da es in JSON HAL aktuell keine Mittel gibt, den in der Datei angegebenen URIs die möglichen HTTP-Methoden zuzuweisen, wird das Format um ein entsprechendes Attribut erweitert. Sollte es sich um eine andere Methode als GET handeln, wird in dem Link-Element das Attribut *method* ergänzt, welches die HTTP-Methode angibt. Trotz dieser Erweiterung bleiben die Dokumente HAL-konform [17, S. 91].

In Listing 11 wird der Aufbau anhand des JSON-HAL-Objektes *element1* aufgezeigt. Über das Link-Element *self* wird die URI hinterlegt, mit der das Objekt aufgerufen wurde. Weiterhin wurden Eigenschaften der Ressource *item1* eingebettet und die URI zur Ressource über den gleichen Namen als Link-Element hinzugefügt.

```
{
  "_links": {
    "self": { "href": "/element1" },
    "item1": { "href": "/item1" },
    "update": { "href": "/element1", "method": "PUT" }
  },
  "_embedded": {
    "item1": {
      "id": "item1",
      "Description": "This is the description of item1"
    }
  },
  "id": "element1",
  "description": "This is the description of element1"
}
```

Listing 11: Beispielhafte Darstellung des JSON-HAL-Objekts Item1 (Quelle: eigene Darstellung)

4.6 Ressourcen- & URI-Design

Dieses Kapitel beschäftigt sich mit dem Aufbau der URIs und der damit verbundenen Identifizierung der Ressourcen.

Zum Ansteuern der API sollte zuerst eine *Root-URI*, also Start-URI festgelegt werden, durch welche die Erkundung der API ermöglicht wird. Zur beispielhaften Darstellung einer Authority wird die URI *example.com* genutzt, welche zur Illustrierung von Beispielen durch die *IANA* (Internet Assigned Numbers Authority) reserviert wurde und zur freien Benutzung steht [87]. Um klarzustellen, dass es sich um eine API handelt, wird folgender Stammpfad genutzt:

```
http://example.com/api...
```

Im Folgenden werden absolute Pfade verwendet, die von diesem Stammpfad ausgehen.

Für das weitere Design der Pfadstruktur müssen die Ressourcen identifiziert werden. Ressourcen in OPC UA sind die Nodes, welche wiederum Attribute haben sowie Referenzen zu anderen Nodes. Ein Ansatz in REST ist es, die hierarchischen Beziehungen auf die Pfadelementstruktur der URI abzubilden [17, S. 48]. Dieser Ansatz eignet sich jedoch nur bedingt für die Abbildung der Referenzen, da OPC UA nicht auf eine bloße Hierarchie beschränkt ist und auch vollvermaschte Strukturen möglich sind [5, S. 140]. Ein weiterer Grund, der gegen die komplette Abbildung der Beziehung zwischen den Nodes auf die Pfadstruktur der URI spricht, ist die Verwendung von stabilen URIs. Das bedeutet, dass URIs, die einmal gültig waren, nicht ins Leere laufen sollen. Dies kann bei einer solchen Abbildung jedoch schnell passieren, da OPC UA ein dynamisches Datenmodell bietet, welches sich stetig ändern kann [5, S. 186]. Aus diesen Gründen wird auf eine hierarchische Abbildung der Referenzen auf das URI-Schema verzichtet.

Bei der Abbildung der Attribute gibt es die Optionen, diese als *State* (Status) in der Node-Ressource anzugeben oder sie als Subressourcen zu modellieren. Die erste Variante, bei welcher die Attribute als State angegeben werden, birgt die Gefahr des Over-fetching, da OPC UA je nach Node-Klasse eine Vielzahl von möglichen Attributen zulässt, die zudem auch unterschiedlich komplex aufgebaut sein können. Ein weiteres Problem stellt das Ändern der Attribute dar. Ein Update für einzelne Attribute würde in dem Fall immer für die gesamte Node-Ressource gelten, da nur diese als Ressource abgebildet ist. Aus diesen Gründen wird die zweite Variante bevorzugt, bei der die Attribute als Subressourcen modelliert werden.

Auf Grund der Eigenschaften von JSON HAL gibt es zwei mögliche Herangehensweisen, die Referenzen und Attribute in Bezug auf die Node abzubilden. Eine Möglichkeit ist es, die Attribute oder Referenzen als eingebettete Elemente (Embedded) zur Node hinzuzufügen. Eine andere Möglichkeit besteht darin, die Referenzen bzw. Attribute als Listenressource in der Node-Ressource zu verknüpfen. Um einen zusätzlichen Aufruf einer Listenressource zu vermeiden, wird auf Grund der begrenzten Menge an Attributen und weil diese direkte Subressourcen der Nodes sind, eine Einbettung in die Node-Ressource bevorzugt. Dies geschieht über Link-Elemente für jedes einzelne Attribut und eine teilweise Abbildung der Attribute als eingebettete Elemente in der Node-Ressource [88, Abschn. 4.1.2]. Auf Grund der endlichen, aber grundsätzlich nicht begrenzten Menge an Referenzen werden diese als Listenressource in die Node eingebunden, womit ein Over-fetching beim Abruf der Node vermieden wird.

Aus diesen Überlegungen ergeben sich die in Tabelle 5 dargestellten Ressourcen, deren URI-Pfade sowie die HTTP-Methoden, die auf die Ressourcen anwendbar sind.

Tabelle 5: Auflistung der Ressourcen (Quelle: eigene Darstellung)

Ressource	Pfad	Methode
Einstiegspunkt	/	GET
Node	/nodes/{nodeid}	GET
Attribut	/nodes/{nodeid}/{attribute}	GET, PUT
Methoden	/nodes/{nodeid}/methods	GET
Methode	/nodes/{nodeid}/methods/{methodid}	GET, POST
Subscription	/nodes/{nodeid}/subscription	GET
Referenzen	/nodes/{nodeid}/references	GET
Referenz	/nodes/{nodeid}/references/{id}	GET
Query	/query	POST

Der Einstiegspunkt stellt eine Ressource dar, die zu der Einstiegs-Node, der Dokumentation und zur Query-Ressource verweist. Die Einstiegs-Node ist eine festgelegte Node, welche ohne weitere Konfiguration die Root-Node darstellt. Da ein OPC UA Server mehrere tausend Nodes haben kann, wird von einer Abbildung aller vorhandenen Nodes als Listenressource abgesehen. Sollte die Nodes-Ressource direkt aufgerufen werden, wird auf die Einstiegs-

Node verwiesen. Die Attribute sind in die Node-Ressource eingebettet inklusive der Pfade zu den einzelnen Attribut-Ressourcen. Bei den Referenzen der Node wird auf die Referenzen-Listenressource verwiesen. In den einzelnen Referenzen befinden sich Verweise auf andere Nodes, deren Pfad wieder dem Muster einer Node-Ressource aus Tabelle 5 folgt. Referenzen werden durch den Referenzentyp sowie die zu verbindenden Nodes definiert, weswegen sie keine Referenz-Id besitzen [5, S. 23]. Aus diesem Grund wird für die einzelnen Referenzen eine automatisch generierte Id zur Adressierung vergeben. Um eine Subscription auf eine Node zu starten, wird die in Tabelle 5 angegebene Ressource aufgerufen. Wenn eine Node Methoden besitzt, werden diese über eine Methoden-Listenressource in der Node-Ressource verlinkt. Die Methoden befinden sich trotz dessen auch in der Ressource Referenzen, da sie Referenzen der entsprechenden Node sind. Die gesonderte Aufführung in der Methoden-Ressource dient für den Bezug zur Node, in deren Kontext eine Methode auf dem OPC UA Server aufgerufen werden soll. Die Ressource Query wird zum Erzeugen und Starten einer Query eingesetzt. Auf Grund der Komplexität und der nicht eindeutigen Zuordnung zu einer der bereits definierten Ressourcen, wird die Ressource direkt im Einstiegspunkt referenziert. Im Anhang C befindet sich eine exemplarische Darstellung der hier angegebenen Ressourcen im Format JSON HAL.

4.7 Abbildung der funktionalen Anforderungen

Die funktionalen Anforderungen stellen typische Anforderung an die Nutzung mit einem OPC UA Server dar. In diesem Teil der Arbeit wird deshalb betrachtet, ob und wie sich die funktionalen Anforderungen mit Hilfe der Funktionen von NodeOPCUA umsetzen lassen.

4.7.1 Verbindungsaufbau zum Server (FA_01 - FA_05)

Die Festlegung des Servers geschieht direkt über den Quellcode der API oder über eine Prozessvariable. Die Verbindung zum OPC UA Server wird bei jeder HTTP-Anfrage initiiert und nach Ausgabe der Ergebnisse wieder geschlossen. Dies hat den Hintergrund, dass OPC UA die Möglichkeit bietet, Nutzern Rechte einzuräumen bzw. dessen Rechte einzuschränken. Da bei dieser Implementierung eine Anmeldung mit Nutzerdaten möglich sein soll (NFA_01), ist es ausgeschlossen eine dauerhafte Session für alle Anfragen zu nutzen. Aus diesem Grund und weil REST eine statuslose Kommunikation bieten soll, wird die Session nach jeder Anfrage geschlossen. Dies bedeutet auch, dass die Authentifizierung bei jedem Verbindungsaufbau durchgeführt werden muss.

4.7.2 Anzeige und Navigation (FA_06 - FA_07)

Eine Durchsuchung des Adressraums, wie in FA_06 beschrieben, wird durch die in OPC UA und NodeOPCUA gleichlautenden Methoden *Browse* und *BrowseNext* ermöglicht. Die Methoden werden für die Bereitstellung der Referenzen-Ressource genutzt. Dabei werden unter Angabe einer NodeId alle vorhandenen Referenzen der Node zurückgegeben. Wird im Pfad keine NodeId angegeben, wird wie bereits beschrieben die Einstiegs-Node für die Browse-Methode genutzt. Dies ermöglicht eine Durchsuchung des Adressraums ohne vorheriges Wissen. BrowseNext wird nicht benötigt, da der OPC UA Client, über den die Web-API auf den OPC UA Server zugreift, keine maximale Anzahl an zurückgegebenen Referenzen pro Node festgelegt hat. Die Markierung bzw. die Aufhebung der Markierung einer Node *(Register-/UnregisterNodes)* wiederspricht der statuslosen Kommunikation in REST und ist auf Grund der geschlossenen Session auch nicht speicherbar [12, S. 42].

4.7.3 Attribute lesen/schreiben (FA_08 - FA_09)

Ein Attribut lässt sich über die NodeOPCUA-Methode *Read* lesen. Der Service benötigt die NodeId der Node sowie das zu lesende Attribut. Zum Ändern einer Attribut-Ressource, wird die HTTP-Methode PUT in Verbindung mit der NodeOPCUA-Methode *Write* genutzt. Dabei wird das Attribut und die Node über den Pfad bestimmt, während sich im übermittelten JSON-Objekt der neue Wert des Attributes befindet.

4.7.4 Historische Werte lesen (FA_10)

Um historische Daten zu lesen, wird die NodeOPCUA-Methode *ReadHistoryValue* benötigt, die neben der NodeId die Angabe einer Start- und Endzeit erfordert. Dies geschieht über die Angabe als Query-Parameter in der URI. Das URI-Template sieht dabei wie folgt aus:

```
http://example.com/api/{nodeid}?start={startTime}&end{endTime}
```

Die Möglichkeit historische Werte zu lesen ist bei einer Variablen nur gegeben, wenn die Attribute *AccessLevel* sowie *UserAccessLevel* den Wert *HistoryRead* aufweisen. Bei Objekten dient das Attribut *EventNotifier* als Indikator. [12, S. 51]

4.7.5 Query (FA_11)

Da eine Query im Gegensatz zu dem Lesen einzelner Attribute wie in 4.7.3 nicht einer einzelnen Ressource zugeordnet werden muss, wird die Möglichkeit geboten, über die Query-Ressource eine neue Query zu starten bzw. zu erzeugen. Die NodeOPCUA-Methode *QueryFirst* nimmt dabei über die HTTP-Methode POST eine Query als JSON-Objekt entgegen, um die Abfrage auszuführen. Der Aufbau der Query entspricht dabei den Spezifikationen von OPC UA für den *QueryFirst Service*[9]. Als Antwort wird das Ergebnis der Query zurückgesendet.

4.7.6 Funktionen aufrufen (FA_12)

Die NodeOPCUA-Funktion *Call* wird genutzt, um eine Methode auf dem OPC UA Server aufzurufen. Die Methode ist in dem Fall die aufgerufene Methoden-Ressource. Über HTTP-POST wird ein JSON-Objekt übermittelt, welches die Parameter für die Methode enthält. Die Notwendigkeit für die Verwendung von POST statt PUT liegt darin begründet, dass OPC-UA-Methoden eine Vielzahl von Operationen ausführen können. Die HTTP-Methode PUT ist idempotent, was bedeutet, dass auch mehrfache Ausführung den gleichen Effekt haben muss wie ein einmaliges Ausführen. Diese Bedingung ist bei dem Aufruf von OPC-UA-Methoden nicht zwingend gegeben, weshalb von einer Verwendung von PUT abgesehen wird.

4.7.7 Subscriptions (FA_13 - FA_16)

Für Subscriptions werden WebSockets genutzt. Um eine WebSocket-Verbindung zu starten, muss eine HTTP-GET-Anfrage gesendet werden, in deren Header nach einem Upgrade auf das WebSocket-Protokoll verlangt wird [89, S. 17]. Diese Anfrage wird vom Server mit dem Statuscode *101: Switching Protocols* beantworten, woraufhin eine WebSocket-Verbindung aufgebaut wird. Dadurch ist eine Verwendung von POST und die damit mögliche Übermittlung von Subscription-Eigenschaften über die HTTP-Anfrage ausgeschlossen. Die Festlegung der Subscription-Eigenschaften wird daher über die dann bestehende WebSocket-Verbindung ermöglicht. Die Subscription wird anfänglich mit vorher definierten Werten gestartet.

Damit eine Subscription zu einer Node gestartet werden kann, muss diese ein *Value*-Attribut aufweisen, also eine Variable sein oder bei dem Attribut *EventNotifier* die Bitmaske für den Wert *SubscribeToEvents* gesetzt haben [12, S. 61].

---

[9] [12, S. 46]

5 Implementierung

In diesem Kapitel wird ein Ansatz für die Umsetzung des Projekts aufgezeigt. Die hier angegebenen Listings sind Auszüge aus dem Code des endgültigen Projekts, die zur Demonstration angepasst bzw. gekürzt wurden.

5.1 Projektstruktur

Die einzelnen Module des Projekts werden in *Controller* und *Router* unterteilt. Die Router dienen zur Weiterleitung der HTTP-Anfragen an die Controller. Die Controller wiederum sind zur Ansteuerung des Informationsmodells von OPC UA und zum Zusammentragen sowie Aufbereiten der Informationen gedacht. [90]

Ausgeführt wird das Projekt über die Datei *app.js*, welche die Module von Express lädt und die Anfragen je nach Route an die entsprechenden Router weiterleitet.

Auf die Nutzung der Befehle *Import* und *Export* zur Einbindung und zum Exportieren der einzelnen Module wurde verzichtet, obwohl diese zu dem im Kapitel *4.1.1 Programmiersprache* genannten Sprachkern gehören. Stattdessen wurde auf die in Node.js üblichen Befehle *Module Exports* und *Require* gesetzt. Dies hat den Hintergrund, dass ältere Versionen von Node.js die Methoden Import und Export nicht unterstützen. Eine solche Version kommt auf dem System der Hochschule Merseburg zum Einsatz.

5.2 Logging und Debugging

Protokollnachrichten auf der Konsole werden über das Tool *debug* ausgegeben, wodurch es möglich ist, Nachrichten bestimmten Namensräumen zuzuordnen und durch unterschiedliche Färbung leichter zu unterscheiden [91]. Dies wird genutzt, um die Nachrichten von WebSockets und REST auseinanderzuhalten. Zusätzlich wird durch das Tool *morgan* eine Anzeige für REST-Verbindungsanfragen für die Ausgabe über debug bereitgestellt. Hierbei wird das Format *dev* genutzt, welches nach Folgendem Schema aufgebaut ist [92]:

```
[Methode]+[URL]+[Status]+[Antwortzeit in Millisekunden]+[Größe der Antwort]
```

Über das Setzen der Umgebungsvariable *debug* kann eingestellt werden, ob diese Ausgaben gewünscht sind.

5.3 Nutzung von Express

Express wird genutzt, um einen HTTP-Server bereitzustellen, welcher die HTTP-Anfragen verarbeitet. Die Nutzung von Express findet dabei in den Projektdateien *app.js* und *router.js* statt.

5.3.1 app.js

Diese Datei stellt das Startskript bzw. die Startdatei dar. Dabei wird neben der Einbindung von Express der Router sowie dessen Route definiert und der Server gestartet. Die Umsetzung wird in einer gekürzten Form im Listing 12 gezeigt.

```
1  const express = require('express');
2  const app = express();
3  const router = require('./routes/router');
4  app.use('/api', router);
5  app.listen(process.env.PORT);
```

Listing 12: Gekürzte Darstellung der Datei app.js (Quelle: eigene Darstellung)

5.3.2 router.js

In dieser Datei werden die Pfade inklusive der HTTP-Methoden aus Kapitel 4.6 modelliert. Eine beispielhafte Darstellung für die Umsetzung einer Route, die mittels der HTTP-Methode GET angefragt wird, ist in Listing 13 gegeben.

```
1  router.get("/route/:id", async (req, res) => {
2      try {
3          res.json(await controller(
4              req.params.id,
5              req.query.start,
6              req.query.end
7          ));
8      } catch (error) {
9          res.status(500).send(error.message)
10     }
11 });
```

Listing 13: Beispielhafte Darstellung einer Anfrage in Express (Quelle: eigene Darstellung)

Die Methode *get* stellt die HTTP-Methode dar. Bei einer anderen Methode, wie bspw. einer POST-Anfrage über denselben Pfad müsste eine gesonderte Funktion geschrieben werden, welche *router.post()* nutzt. Der Funktionsparameter *req* aus der Callback-Methode bietet den Zugriff auf eine Vielzahl von Werten aus der HTTP-Anfrage. Wichtig für dieses Projekt sind die Parameter, die Queries, der Body und der Header. Verfügbare Parameter werden in Express über den Pfad mit einem führenden Doppelpunkt definiert, wie im Listing 13 beim Parameter *:id* zu sehen ist. Da Fragezeichen laut OPC UA erlaubte Zeichen in NodeIds sind, müssen die NodeIds mittels URL-Kodierung in der Adresszeile angegeben werden [12, S. 166], [93, S. 65]. Über den Body werden bei den Methoden PUT und POST die Nutzdaten übertragen.

Die Antwort auf die HTTP-Anfrage wird über den Funktionsparameter *res* in Form einer JSON-Datei zurückgegeben. Im Catch-Block werden Fehler abgefangen. Je nach Fehler wird ein passender HTTP-Statuscode gesendet, zusammen mit einer individuellen Nachricht. Dies geschieht im fertigen Code über den Abgleich des Strings *error.message*.

5.3.3 express-ws

Bei express-ws handelt es sich um ein Paket zur Einbindung von WebSocket-Endpunkten in Express. Somit ist eine Bereitstellung von WebSockets über Express-Routen möglich.

```
// app.js

const expressWs = require('express-ws')(app);

// router.js

router.ws("/route/:id/subscription", (ws, req) => {
    try {
        const sub = subscription(ws, req.params.id);

        ws.on('close', function () {
            debug("WebSocket closed")
        })
    } catch (error) {
        debug(error.message)
    }
})
```

Listing 14: Beispielhafte Darstellung der Einbindung von WebSockets (Quelle: eigene Darstellung)

Wie in Listing 14 zu sehen ist, bietet express-ws eine ähnliche Implementierung, wie es in Express üblich ist. Die Callback-Funktion bietet hierbei zusätzlich den Funktionsparameter *ws*, welcher eine Instanz der WebSocket-Klasse aus der WebSocket-Bibliothek *ws* bereitstellt [94]. Dies ermöglicht die Nutzung von *Event Handler* (Ereignisbehandlungsroutinen) in Bezug auf die bestehende WebSocket-Verbindung. Im Projekt wird so zum Beispiel eine Nachricht bei aktiviertem Debugging (siehe 5.2) ausgegeben, die bei dem Event *Close* eine Nachricht

sendet, dass die WebSocket-Verbindung geschlossen wurde. Ein weiteres Event stellt *Message* dar, dass bei einer eingehenden Nachricht erzeugt wird und die empfangene Nachricht zur Verfügung stellt. Über die Methode *ws.send* können Nachrichten an den verbundenen Client gesendet werden.

5.3.4 Basic Auth

Wie bereits in 4.7.1 beschrieben, ist bei jeder Anfrage eine Authentifizierung notwendig. Dies geschieht mittels HTTP-Authentifizierung (siehe 4.4). Um das zu ermöglichen wird der *Header* des Funktionsparameters *req* genutzt, um an die unter dem Parameter *Authorization* hinterlegten Informationen zu gelangen *(req.headers.authorization)*. Der in Base64 kodierte Benutzername und Passwort werden dekodiert und bei jedem Aufruf der Controller als zusätzliche Parameter übertragen.

5.4 Controller

Die Controller sind dazu gedacht, die Informationen der Ressource zusammenzutragen und diese an die sie aufrufende Express-Route zurückzugeben.

```
1  module.exports = async function exampleController(nodeId, login, password) {
2      try {
3          const userIdentity = login === "" || password === "" ? null : {
4              type: UserTokenType.UserName,
5              userName: login,
6              password: password
7          };
8          const client = await connect();
9          const session = await client.createSession(userIdentity);
10
11         const result = await session.browse(nodeId);
12         if (result.statusCode !== StatusCodes.Good) {
13             await session.close();
14             await client.disconnect();
15             throw new Error(result.statusCode.toString());
16         }
17         await session.close();
18         await client.disconnect();
19
20         return result.toJSON();
21     } catch (err) { throw new Error(err.message); }
22 }
```

Listing 15: Beispiel-Controller (Quelle: eigene Darstellung)

Im Code aus Listing 15 wurde ein Beispielcontroller modelliert, der den Umgang mit NodeOPCUA aufzeigt. Die Methode *Client* wird dabei aus dem Modul *client-connect.js* importiert, welche die Umgebungsvariablen *opcuaServerUrl* sowie *opcuaLocal* nutzt, um eine Instanz eines OPCUA-Clients zu erzeugen. Die Umgebungsvariable opcuaLocal wird benötigt,

wenn die API und der OPC UA Server auf dem gleichen Gerät ausgeführt werden. Mit der Client-Instanz und den Nutzerdaten wird anschließend eine Session erzeugt, über die OPC UA Services gestartet werden können. In diesem Beispielcontroller wird exemplarisch der Browse Service verwendet. Jedes Ergebnis eines Serviceaufrufs in OPC UA liefert einen Statuscode zurück. Über die Prüfung dieses Statuscodes kann eine Fehlerbehandlung stattfinden, was in diesem Fall eine Weitergabe des Fehlercodes an den Router bedeutet, der die Fehlercodes abgleicht und entsprechende HTTP-Fehler ausgibt. Nach der Ausführung des Services wird die Session sowie die Client-Instanz geschlossen und das Ergebnis als JSON-Objekt an den Router zurückgegeben. Die im Beispiel verwendeten Methoden *toString()* und *toJSON()* sind Methoden aus NodeOPCUA, die eine bessere Darstellung der Daten bieten, als dies bei den eingebauten Methoden von JavaScript der Fall ist.

Im Folgenden wird eine Auswahl der Controller genannt und es wird ein kurzer Überblick über die Funktionen der Controller gegeben. Auf eine Darstellung als Listing wird dabei wenn möglich verzichtet, da sich der gesamte Code mit Erläuterungen im Anhang D finden lässt.

5.4.1 AttributeDetail.js

Dieser Hilfscontroller stellt ein Modul dar, dass von mehreren anderen Controllern genutzt wird, um eine geeignete Darstellung der Attribute zu bieten. Die meisten Attribute werden durch NodeOPCUA nur in kurzer Form wiedergegeben. Ein Beispiel dafür ist der EventNotifier, welcher eine Bitmaske wiedergibt, bei der das erste Bit gesetzt sein muss, damit zu Events dieser Node eine Subscription gestartet werden kann. Statt einer Bitmaske, die als einfache Dezimalzahl übermittelt wird, gibt dieses Modul ein Array zurück, dass die gesetzten Eigenschaften der Bitmaske aufzeigt. Sollten zum Beispiel alle 3 Bits der Maske gesetzt sein, so würde das JSON-Objekt das folgende Attribut enthalten:

```
EventNotifier: [SubscribeToEvents, HistoryRead, HistoryWrite]
```

Der Controller gibt dabei nicht selbst das JSON-Objekt in JSON-HAL an den Router zurück, sondern den Wert des Attributs an den Controller, der diesen Hilfscontroller aufgerufen hat. Die Grundlage für die Umsetzung der Methoden für die einzelnen Attribute sind die OPC-UA-Spezifikationen.

In der aktuellen Implementierung wurden die Attribute *InverseName*, *ArrayDimensions*, *RolePermissions* und *UserRolePermissions* noch nicht umgesetzt. Diese können wie die bereits umgesetzten Attribute anhand der OPC-UA-Spezifikationen realisiert werden [93].

5.4.2 getPossibleAttributes.js

Hierbei handelt es sich um einen weiteren Hilfscontroller, der für die Erzeugung der möglichen Link-Elemente der Attribute im JSON-HAL-Objekt zuständig ist. Weiterhin stellt der Controller eine Auswahl von Standardattributen bereit, die als eine Art Voransicht für die Node-Ressource bereitgestellt werden. Dafür werden alle in OPC UA möglichen Attribute als Array in den Read Service gegeben. Anschließend wird der OPC-UA-Statuscode für jedes zurückgegebene Attribut geprüft. Sollte ein Attribut nicht vorhanden sein, wird der Statuscode *BadAttributeIdInvalid* zurückgegeben [12, S. 51]. In diesem Fall wird das Attribut nicht dem JSON-Objekt hinzugefügt. Bei den Standardattributen handelt es sich um die NodeId, NodeClass, BrowseName, DisplayName und Description.

5.4.3 getNode.js

Mit Hilfe dieses Controllers wird die Ansicht der Node-Ressource modelliert. Um zu prüfen, ob die Node Referenzen oder Methoden aufweist, wird der Browse Service genutzt. Sollte einer der Fälle zutreffen, wird das JSON-HAL-Objekt um die entsprechenden Link-Elemente zu den Listenressourcen erweitert. Wenn die Node die Option bieten, historische Werte auszulesen oder die Möglichkeit einer Subscription aufweist, dann werden die entsprechenden Link-Elemente in der Ausgabe ergänzt.

Der Hilfscontroller *getPossibleAttributes.js* erweitert die Link-Elemente um die URIs der verfügbaren Attribute und fügt die Standardattribute als eingebettete Elemente hinzu.

5.4.4 callMethod.js & getMethod.js

Über den Controller *callMethod* wird der Methodenaufruf umgesetzt. Um dies zu ermöglichen, wird zuerst über den Browse Service die NodeId der Eingabeparameter *(InputArguments)* für die Funktion abgefragt. Anschließend liefert der Read Service die Eigenschaften, also das Schema der Parameter. Anhand dieses Schemas können die per HTTP-POST übermittelten Nutzdaten abgeglichen werden. Das Schema kann mit der HTTP-GET-Methode über die gleiche URL abgerufen werden. Nach der Prüfung der übermittelten Daten wird damit der Call Service aufgerufen und das Ergebnis zurückgegeben.

Der Controller *getMethod* wird auf eine ähnliche Art und Weise umgesetzt, nur das dieser eine Ansicht der Eingabe- und Ausgabeparameter, sowie die Standartattribute aus dem Hilfscontroller *getPossibleAttributes.js* bereitstellt.

5.4.5 getAttribute.js

Bei der URI für den Aufruf eines Attributes ist die Angabe des Attributnamens als Zahl oder als String möglich, wobei für die Angabe der Links in den JSON-HAL-Objekten Strings genutzt werden. Um den Aufruf nicht vorhandener Attribute abzufangen, wird der Statuscode des Read Services geprüft. Die Ergebnisse aus dem Read Service werden als Objekt unter dem Namen Value in das JSON-HAL-Objekt eingefügt. Zusätzlich wird über den Hilfscontroller *AttributeDetails.js*, falls vorhanden, eine aufbereitete Ansicht des Attributes über dessen Attributnamen im JSON-HAL-Objekt bereitgestellt.

Um zu prüfen, ob das Attribut veränderbar ist, wird die Bitmaske des Attributs *UserWriteMask* abgeglichen. Sollte es sich um das Attribut *Value* handeln, wird hingegen die Bitmaske des Attributes *UserAccessLevel* geprüft. Auf eine Prüfung von *WriteMask* und *AccessLevel* kann in diesem Fall verzichtet werden, da diese mindestens die gleichen Restriktionen aufweisen wie die beiden nutzerspezifischen Bitmasken. Sollte das Attribut schreibbar sein, wird ein entsprechender Link unter dem Namen *Update* in der Ausgabe ergänzt. [93, S. 17,30]

5.4.6 writeAttribute.js

Dieser Controller wurde noch nicht vollständig implementiert. Aktuell ist der Controller dazu in der Lage, eine *NodeId*, *AttributeId* und den *DataValue* entgegenzunehmen und diese mit Hilfe des Write Service zu verarbeiten. Der Parameter *IndexRange* wurde bisher noch nicht implementiert und dient dazu, um bei einem vorhandenen Array von Werten den zutreffenden Index für die neuen Daten festzulegen [12, S. 55]. Da bisher nur ein DataValue unterstützt wird, wird dieser als einzelner Wert in einem Array übergeben. Der DataValue wiederum besteht aus verschiedenen Komponenten, wie einem Statuscode, Zeitstempeln und den eigentlichen Wert [12, S. 128]. Hier wird aktuell die Übergabe von *Value* unterstützt, ohne dessen Komponente *arrayType* [95]. Alle Werte, die noch nicht implementiert wurden, sind optional unter der Bedingung, dass es sich um einen einzelnen skalaren Wert handelt. Somit ist der Controller zwar grundsätzlich funktionsfähig, kann aber nicht mit allen Eventualitäten umgehen. Entsprechende To-do-Kommentare für die fehlenden Werte inklusive der für die Umsetzung notwendigen Dokumentationen wurden im Quelltext hinterlegt.

Da bei einer solchen Operation viele Fehler auftreten können, wurde die komplette Prüfung über die für diese Operation möglichen OPC-UA-Statuscodes gelöst, welche alle in HTTP-Statuscodes umgesetzt wurden [12, S. 56].

Da der Write Service nur einen Statuscode zurückgibt, wird für die Ausgabe der HTTP-Statuscode 204 als Antwort zurückgegeben, welcher keinen Nachrichtenkörper enthält.

5.4.7 getHistory.js

Dieser Controller wird über den gleichen Router-Pfad aufgerufen wie der Controller *getNode.js*. Der Controller wird genutzt, sobald der Query-Parameter *start* mit oder ohne den Query-Parameter *end* gesetzt wurden. Diese Parameter geben einen Zeitraum für das Lesen der historischen Werte vor.

Sollte keine Endzeit angegeben sein, wird die aktuelle Zeit genutzt, sodass alle Werte von der Startzeit bis zum Zeitpunkt der Anfrage ausgegeben werden. Der Controller benutzt den Konstruktor für ein JavaScript *Date*-Objekt, um möglichst viele Angaben von Zeitformaten zu unterstützen [96]. Nach einer Prüfung, ob die Node *HistoryRead* unterstützt, wird der History Read Service dazu genutzt, um die Daten für den angegeben Zeitraum zu gewinnen. Diese werden als Array über das Attribut *HistoryData* in dem JSON-HAL-Objekt ausgegeben.

5.4.8 getReferences.js & getMethods.js

Diese beiden Controller werden dazu eingesetzt, um die Listenressourcen der Referenzen bzw. Methoden bereitzustellen. Der Controller *getReferences.js* gibt dabei die gesamten Referenzen einer Node zurück und bildet so die Beziehungen zwischen den Nodes ab, wodurch ein Erkunden des Adressraums ermöglicht wird.

Neben einem Link-Element für jede Referenz, wird auch jeweils ein Embedded-Element bereitgestellt, welches die NodeId der referenzierten Node, den Referenztyp und die Referenzrichtung *(BrowseDirection)* bereithält. Die dafür notwendigen Daten werden durch den Browse Service gewonnen, während der Read Service dazu genutzt wird, statt der NodeId des Referenztyps den Anzeigenamen *(DisplayName)* bereitzustellen.

Ähnlich wurde der Controller *getMethods.js* umgesetzt, welcher die verfügbaren Methoden einer Node ausgibt. Statt des Referenztyps und der Referenzrichtung wird für jede Methode zusätzlich zu den bei einer Referenz angegebenen Werten auch der Anzeigename und die Beschreibung *(Description)* der Methode angezeigt.

5.4.9 getReference.js

Der Controller gibt eine ausführliche Ansicht einer Referenz wieder. Es wird lediglich der Browse Service genutzt, um die Daten der referenzierten Node und des Referenztyps anzuzeigen. Für die Node und den Referenztyp werden die NodeId sowie der Anzeigename ausgegeben. Das Objekt der Node wird zudem um das Attribut der Node-Klasse erweitert. Weiterhin wurden Link-Elemente zum Referenztyp und der referenzierten Node eingefügt.

5.4.10 Subscription.js

Mit Hilfe von *express.ws* wird, wie in 5.3.3 beschrieben, eine WebSocket-Instanz bereitgestellt und an den Controller *subscription.js* übergeben. Nach einer vorherigen Prüfung, ob die Node über das Attribut *Value* verfügt oder über eine entsprechend gesetzte Bitmaske im Attribut *EventNotifier* (*Bit 0: SubscribeToEvents*) wird eine Subscription initiiert, welche je nach Node entweder den EventNotifier oder den Value-Wert überwacht. Sollte eine Änderung festgestellt werden, wird die WebSocket-Instanz genutzt, um eine WebSocket-Nachricht mit den entsprechenden Werten an den Client zu senden.

Im Falle eines EventNotifiers wird ein Filter für die Überwachungsoptionen *(monitoringOptions)* genutzt. Dieser Filter gibt an, welche Felder im Falle eines Ereignisses ausgegeben werden. Aktuell wird der Event-Typ, die NodeId, der Anzeigename der Quell-Node, die Zeit und die Nachricht des Ereignisses ausgegeben.

Die Möglichkeit die Eigenschaften der Subscription und die Überwachungsoptionen durch den Client ändern zu lassen, muss noch implementiert werden. Dies wäre zum Beispiel durch eine WebSocket-Nachricht des Clients möglich, deren Werte nach einer Validierung für die Subscription übernommen werden. Aktuell sind hierfür nur vordefinierte Werte angegeben. Entspricht diese Nachricht nicht dem geforderten Schema, sollte eine Hilfsnachricht mit dem Schema an den Client gesendet werden.

5.4.11 Queries

Queries wurden für diese Implementierung auf Grund ihrer hohen Komplexität noch nicht integriert [12, S. 44]. Die mögliche Vorgehensweise orientiert sich dabei an den OPC-UA-Spezifikationen zum *QueryFirst* Service [12, Kap. 5.9.3]. Über die HTTP-POST-Methode werden die Anfrageparameter für diesen Service als JSON-Objekt übermittelt. Diese Anfrageparameter müssen anschließend validiert werden, soweit dies auf Grund der Komplexität möglich ist. Für die Fehlerausgabe sollten alle in der Spezifikation des *QueryFirst* Service genannten Statuscodes als HTTP-Statuscodes umgesetzt werden.

6 Ergebnisse und Ausblick

6.1 Beschreibung und Bewertung der Vorgehensweise

Zuerst wurde eine systematische Analyse vorgenommen, bei der anhand des Ist- & Soll-zustandes Anwendungsfälle formuliert wurden. Aus diesen Anwendungsfällen wurden die Anforderungen an die Web-API aufgestellt. Um diese Anforderungen zu erfüllen, fand eine Analyse darüber statt, welche Programmiersprache und Frameworks sich am besten zur Umsetzung des Projekts eignen. Daraufhin wurden die verschiedenen Möglichkeiten der Umsetzung untersucht. Durch diese Untersuchungen war es möglich, die am besten geeigneten Mittel und Vorgehensweisen zur Realisierung auszuwählen und das Projekt entsprechend zu Implementieren.

6.2 Zusammenfassung

Die zu Beginn dieser Arbeit aufgestellte Anforderung und Zielsetzung wurde durch die hier vorgestellte Umsetzung erfüllt. Mit dieser API ist ein Nutzer dazu in der Lage, auf einen OPC UA Server zuzugreifen, ohne dabei direkt einen OPC UA Client einsetzen zu müssen. Dem Nutzer wird es somit ermöglicht, den Adressraum eines OPC UA Servers zu durchsuchen und dessen Attribute abzufragen. Weiterhin erlaubt die API eine Bearbeitung dieser Attribute, das Aufrufen von Methoden, das Starten einer OPC UA Query und das Erstellen von Subscriptions, wodurch eine automatische Zusendung der abonnierten Werte ermöglicht wird. Die Webtechnologien, die dabei eingesetzt werden, sind WebSockets für Subscriptions und REST für die anderen Abfragen. Somit werden moderne Webtechnologien genutzt, um die Anforderungen umzusetzen.

6.3 Ausblick & Fazit

Wie bereits in den Unterpunkten von Kapitel 5.4 aufgezeigt wurde, handelt es sich bei der aktuellen Umsetzung nicht um eine vollständige Implementierung. Neben der in 5.4.1 genannten fehlenden Ansicht einiger Attribute, bedarf der Controller *writeAttribute.js* einer umfassenderen Umsetzung, wodurch auch komplexe Werte bzw. Strukturen von Variablen geändert werden können. Auch eine Möglichkeit zur Änderung der Subscription-Eigenschaften, wie in 5.4.10 beschrieben, sollte bei einer Weiterführung dieses Projekts berücksichtig werden. Eine weitere sehr komplexe Aufgabe stellt die Umsetzung der Query-Ressource über einen entsprechenden Controller dar. Für große und stark verzweigte

Adressräume eignet sich zudem eine Umsetzung der Ausgabe der Controller getReferences.js und getMethods.js mittels Paginierung [17, S. 38]. Auch im Bereich Authentifizierung bietet diese Implementierung bisher keine vollständige Umsetzung, damit sie mit einer größeren Zahl an OPC-UA-Server-Konfigurationen umgehen kann. Aktuell wird nur die Authentifizierung durch einen Nutzernamen und Passwort angeboten, während NodeOPCUA auch die Nutzung von X.509-Zertifikaten unterstützt. Um die API einem breiten Umfeld zur Verfügung zu stellen, bedarf es außerdem einer ausführlicheren Dokumentation bzw. Beschreibung der API zum Beispiel durch Werkzeuge wie *Swagger*, das die Möglichkeit zur Erstellung einer Schnittstellenbeschreibungen bietet [97].

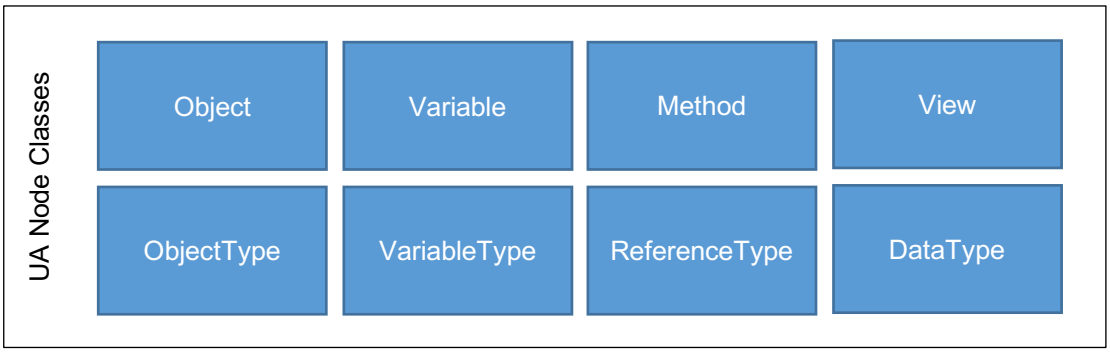
Ein persönlicher Anspruch, dessen Umsetzung auf Grund der begrenzten Zeit leider keine Beachtung finden konnte, ist eine Unterstützung für das Ansteuern verschiedener OPC UA Server über ein und dieselbe Schnittstelle. Dies bietet den Vorteil, dass nicht für jeden OPC UA Server eine eigene Instanz der Web-API laufen muss. Realisiert werden könnte das über eine Startseite, welche die verschiedenen Endpunkte der OPC UA Server als JSON-Objekt zurückgibt. Die Endpunkte liefern dabei neben einer URI auch Informationen zu den geforderten Sicherheitseinstellungen und könnten über entsprechende Sub-Pfade unterhalb der genannten Startseite realisiert werden. Eine Liste und die Informationen der Endpunkte könnte hierbei über eine Datenbank bezogen werden oder über einen *Discovery Server,* welcher die Informationen der verschiedenen OPC UA Server bereitstellt [98].

Bei der Recherche fiel auf, dass viele OPC UA SDKs die Spezifikationen nicht vollständig unterstützen. Die Spezifikationen wurden sehr umfänglich formuliert und gerade SDKs von kleineren Entwicklern weisen noch Lücken bei der kompletten Unterstützung auf. So war es sehr ärgerlich während der Umsetzung festzustellen, dass NodeOPCUA keine Token-basierte Authentifizierung unterstützt. Auch die Query Services sind laut Herstellerseite noch nicht vollständig implementiert, wobei nicht genauer erläutert wird, welche Funktionen fehlen. Auch die Unterstützung des *NodeManagement Service Set* und *HistoryUpdate* fehlt gänzlich. [99]

Dies zeigt unteranderem die hohe Komplexität von OPC UA auf, die sich zukünftig durchaus auch als Nachteil für die Unterstützung der Technologie herausstellen könnte. Das MQTT-Protokoll, welches sich selbst als Standard für IoT bezeichnet, bietet hier zum Beispiel einen wesentlich leichteren Einstieg bei ebenfalls sehr großen Funktionsumfang. Mehr als die Hälfte der Mitglieder der OPC Foundation kommt zudem aus dem europäischen Raum, während eine technologische Weltmacht wie China mit gerade mal 8 % einen recht kleinen Anteil darstellt. Es ist wichtig, dass sich OPC UA den technologischen Anforderungen der Welt anpasst, damit es mit anderen Technologien Schritt halten kann und weiterhin relevant ist. [1, S. 19], [100]

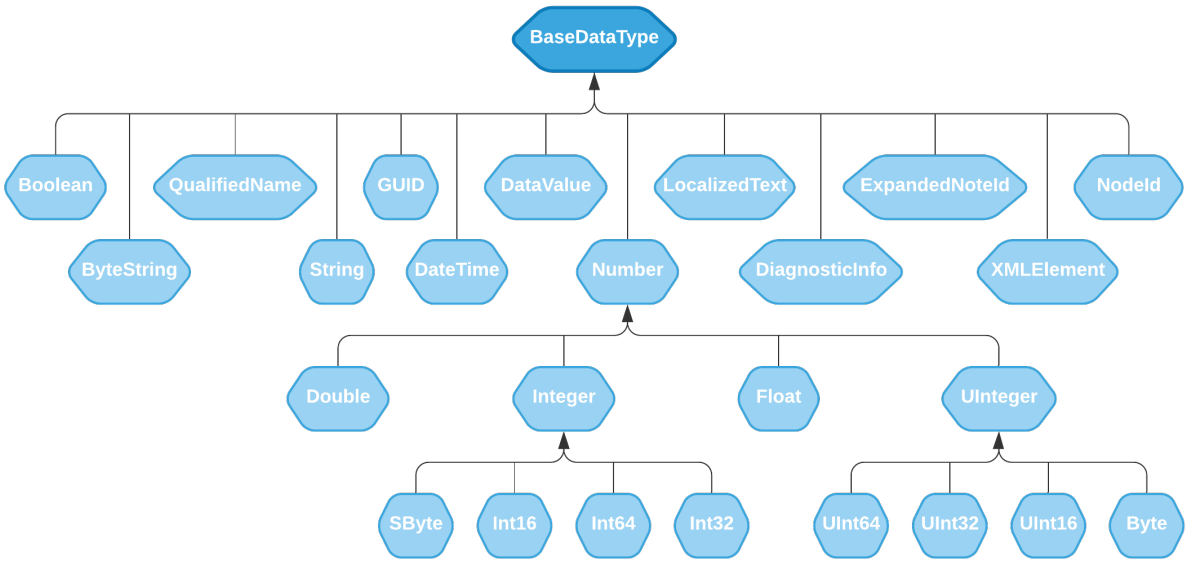
Anhang

A Übersicht der UA-Node-Klassen



(Quelle: Eigene Darstellung)

B Hierarchische Darstellung der eingebauten Datentypen



(Quelle: Eigene Darstellung in Anlehnung an [5, S. 63])

C Exemplarische Abbildung der Ressourcen in JSON-HAL

C.1 Node

```
GET /nodes/:nodeId

{
  "_links": {
    "self": { "href": "/nodes/:nodeId" },
    "subscription": { "href": "/nodes/:nodeId/subscription" },
    "references": { "href": "/nodes/:nodeId/references" },
    "Methods": {"href":  "/nodes/:nodeId/methods"},
    "NodeId": { "href": "/nodes/:nodeId/nodeid" },
    "NodeClass": { "href": "/nodes/:nodeId/nodeclass" },
    "BrowseName": { "href": "/nodes/:nodeId/browsename" },
    "DisplayName": { "href": "/nodes/:nodeId/displayname" },
    "Description": { "href": "/nodes/:nodeId/description" },
    "WriteMask": { "href": "/nodes/:nodeId/writemask" },
    "UserWriteMask": { "href": "/nodes/:nodeId/userwritemask" },
    "Value": { "href": "/nodes/:nodeId/value" },
    "DataType": { "href": "/nodes/:nodeId/datatype" },
    "ValueRank": { "href": "/nodes/:nodeId/valuerank" },
    "ArrayDimensions": { "href": "/nodes/:nodeId/arraydimensions" },
    "AccessLevel": { "href": "/nodes/:nodeId/accesslevel" },
    "UserAccessLevel": { "href": "/nodes/:nodeId/useraccesslevel" },
    "MinimumSamplingInterval": { "href": "/nodes/:nodeId/minimumsamplinginterval" },
    "Historizing": { "href": "/nodes/:nodeId/historizing" }
  },
  "_embedded": {
    "NodeId": ":nodeId",
    "NodeClass": "Variable",
    "BrowseName": "ExampleNode",
    "DisplayName": "ExampleNode",
    "Description": "This is an example"
  }
}
```

(Quelle: eigene Darstellung)

C.2 Attribut

```
GET /nodes/:nodeId/value

{
  "_links": {
    "self": {
      "href": "/nodes/:nodeId/Value"
    },
    "update": {
      "href": "/nodes/:nodeId/value",
      "method": ["PUT"]
    }
  },
  "value": {
    "dataType": "Double",
    "arrayType": "Scalar",
    "value": 48
  },
  "time": {
    "sourceTimestamp": "2021-10-17T10:11:06.383Z",
    "sourcePicoseconds": 0,
    "serverTimestamp": "2021-10-17T10:11:07.370Z",
    "serverPicoseconds": 0
  }
}
```

Quelle: eigene Darstellung

C.3 Referenzen

```
GET /nodes/:nodeId/references

{
  "_links": {
    "self": { "href": "/nodes/:nodeId/references" },
    "1": { "href": "/nodes/:nodeId/references/1" },
    "2": { "href": "/nodes/:nodeId/references/2" }
  },
  "_embedded": {
    "1": {
      "NodeId": "i=63",
      "ReferenceType": "HasTypeDefinition"
    },
    "2": {
      "NodeId": "/:nodeId2",
      "ReferenceType": "HasComponent"
    }
  }
}
```

Quelle: eigene Darstellung

C.4 Referenz

```
GET /nodes/:nodeId/references/1

{
  "_links": {
    "self": { "href": "/nodes/:nodeId/references/1" },
    "Node": { "href": "/nodes/i%3D63" },
    "ReferenceType": { "href": "/nodes/i%3D40" }
  },
  "_embedded": {
    "Node": {
      "nodeId": "i=63",
      "DisplayName": "BaseDataVariableType",
      "NodeClass": "Variable"
    },
    "ReferenceType": {
      "nodeId": "i=40",
      "DisplayName": "HasTypeDefinition"
    }
  },
  "isForward": true
}
```

Quelle: eigene Darstellung

C.5 Methoden

```
GET /nodes/:nodeId/methods

{
  "_links": {
    "self": { "href": "/nodes/:nodeId/methods" },
    ":methodId1": { "href": "/nodes/:nodeId/methods/:methodId1" },
    ":methodId2": { "href": "/nodes/:nodeId/methods/:methodId2" }
  },
  "_embedded": {
    ":methodId1": {
      "NodeId": ":methodId1",
      "DisplayName": "methodId1",
      "Description": "This is Method 1."
      },
    ":methodId2": {
      "NodeId": ":methodId2",
      "DisplayName": "methodId2",
      "Description": "This is Method 2."
    }
  }
}
```

Quelle: eigene Darstellung

C.6 Methode

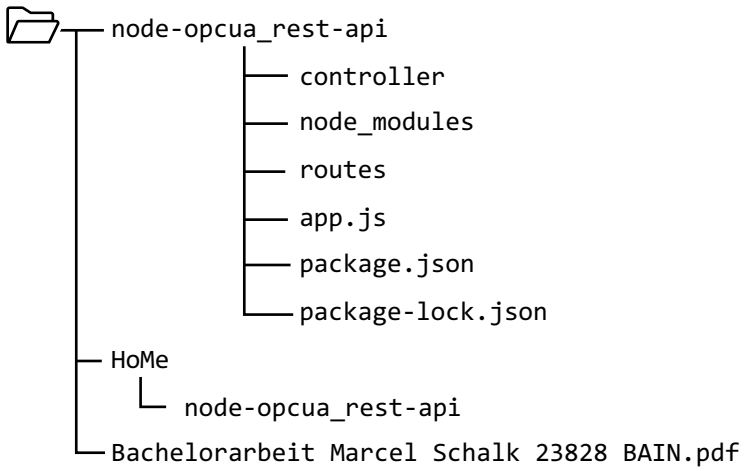
```
GET /nodes/:nodeId/methods/:methodId

{
  "_links": {
    "self": { "href": "/nodes/:nodeId/methods/:methodId" },
    "call": { "href": "/nodes/:nodeId/methods/:methodId", "method": "POST" },
    "inputArguments": {"href": "/nodes/:nodeIdInputArguments"},
    "outputArguments": {"href": "/nodes/:nodeIdOutputArguments"}
  },
  "_embedded": {
    "inputArguments": {
      "Values": {},
      "ValueRank": {},
      "ArrayDimensions": {}
    },
    "OutputArguments": {
      "Values": {},
      "ValueRank": {},
      "ArrayDimensions": {}
    }
  },
  "NodeId": ":methodId",
  "NodeClass": "Method",
  "BrowseName": "ExampleMethod",
  "DisplayName": "ExampleMethod",
  "Description": "This is an example Method"
}
```

Quelle: eigene Darstellung

D Begleit-DVD

Auf der Begleit-DVD befindet sich neben einer Kopie der Bachelorarbeit der Quellcode der Web-API. Zusätzlich ist im Ordner *HoMe* eine angepasste Version der API zu finden, die auf HTTP-Authentifizierung und Umgebungsvariablen verzichtet. Auf Grund der Menge an Dateien beinhaltet diese Verzeichnisvorschau nur die Daten bis zur Verzeichnisebene der Tiefe zwei.



7 Literaturverzeichnis

[1] OPCFoundation, „OPC-UA-Interoperability-For-Industrie4-and-IoT-EN.pdf“, Juni 2020. https://opcfoundation.org/wp-content/uploads/2017/11/OPC-UA-Interoperability-For-Industrie4-and-IoT-EN.pdf (zugegriffen Juli 14, 2021).

[2] R. Armstrong, „OPC 10000-6 - UA Specification Part 6 - Mappings 1.04“.

[3] „OPC Unified Architecture“, *Wikipedia*. Jan. 20, 2021. Zugegriffen: Juli 24, 2021. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=OPC_Unified_Architecture&oldid=207860035

[4] „OPC Foundation“, *Wikipedia*. Apr. 14, 2021. Zugegriffen: Juli 24, 2021. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=OPC_Foundation&oldid=1017834777

[5] W. Mahnke, S.-H. Leitner, und M. Damm, *OPC unified architecture*. Berlin: Springer, 2009.

[6] J. Lange, F. Iwanitz, und T. J. Burke, *OPC: von Data Access bis Unified Architecture*, 5., Durchges. Aufl. Berlin Offenbach: VDE-Verl, 2014.

[7] „History“, *OPC Foundation*. https://opcfoundation.org/about/opc-foundation/history/ (zugegriffen Juli 26, 2021).

[8] OPCFoundation, „OPC Unified Architecture - Part 3: Address Space Model“.

[9] OPCFoundation, „Core Data Types Argument“, *OPC UA Online Reference*. https://reference.opcfoundation.org/v104/Core/DataTypes/Argument/ (zugegriffen Aug. 12, 2021).

[10] „High Performance OPC UA Server SDK: OPC UA Namespaces“. https://documentation.unified-automation.com/uasdkhp/1.5.2/html/_l2_ua_adr_space_concept_namespaces.html (zugegriffen Aug. 13, 2021).

[11] J. Harding und K. Deiretsbacher, „OPC Unified Architecture Part 100: Devices“.

[12] OPCFoundation, „OPC 10000-4 - UA Specification Part 4: Services“.

[13] OPCFoundation, „OPC 10000-7 Unified Architecture Part 7 Profiles“, *OPC UA Online Reference*. https://reference.opcfoundation.org/v104/Core/docs/Part7/ (zugegriffen Aug. 15, 2021).

[14] D. Kress, *GraphQL: eine Einführung in APIs mit GraphQL*, 1. Auflage. Heidelberg: dpunkt.verlag, 2021.

[15] „API“, *Wikipedia*. Okt. 11, 2021. Zugegriffen: Okt. 20, 2021. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=API&oldid=1049339887

[16] „Hypertext Transfer Protocol“, *Wikipedia*. Okt. 19, 2021. Zugegriffen: Okt. 20, 2021. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=Hypertext_Transfer_Protocol&oldid=1050794358

[17] S. Tilkov, Hrsg., *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*, 3., Aktualisierte und erw. Aufl. Heidelberg: dpunkt-Verl, 2015.

[18] „rfc3986“. https://datatracker.ietf.org/doc/html/rfc3986 (zugegriffen Okt. 20, 2021).

[19] „HTTP Messages - HTTP | MDN“. https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages (zugegriffen Okt. 20, 2021).

[20] „HTTP/1.1: HTTP Message“. https://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html (zugegriffen Okt. 20, 2021).

[21] „HTTP request methods - HTTP | MDN“. https://developer.mozilla.org/de/docs/Web/HTTP/Methods (zugegriffen Okt. 24, 2021).

[22] „Safe (HTTP Methods) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN“. https://developer.mozilla.org/en-US/docs/Glossary/Safe/HTTP (zugegriffen Okt. 24, 2021).

[23] „Idempotent - MDN Web Docs Glossary: Definitions of Web-related terms | MDN“. https://developer.mozilla.org/en-US/docs/Glossary/Idempotent (zugegriffen Okt. 24, 2021).

[24] „Cacheable - MDN Web Docs Glossary: Definitions of Web-related terms | MDN“. https://developer.mozilla.org/en-US/docs/Glossary/cacheable (zugegriffen Okt. 24, 2021).

[25] „HTTP response status codes - HTTP | MDN“. https://developer.mozilla.org/de/docs/Web/HTTP/Status (zugegriffen Okt. 24, 2021).

[26] „HTTP-Statuscode“, *Wikipedia*. Juni 03, 2021. Zugegriffen: Okt. 24, 2021. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=HTTP-Statuscode&oldid=212649723

[27] „Transmission Control Protocol“, *Wikipedia*. Juli 21, 2021. Zugegriffen: Aug. 18, 2021. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=Transmission_Control_Protocol&oldid=214090199

[28] Prosys OPC Ltd, Hrsg., „Prosys OPC UA Simulation Server Manual“. Mai 28, 2021.

[29] ascolab GmbH, „Whitepaper: Overview OPC UA“. Sep. 2010.

[30] W. Jackson, *JSON quick syntax reference*. Lompoc, CA: Apress, 2016.

[31] „JSON“, *Wikipedia*. Aug. 18, 2021. Zugegriffen: Aug. 18, 2021. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=JSON&oldid=1039443909

[32] Helmut Balzert, *Lehrbuch der Software-Technik. 1: Software-Entwicklung*, 2. Aufl., 1. Nachdr. Heidelberg Berlin: Spektrum, 2001.

[33] „How to issue HistoryUpdate Service - Unified Automation Forum“. https://forum.unified-automation.com/viewtopic.php?f=22&t=7573 (zugegriffen Sep. 27, 2021).

[34] Prosys, „Prosys OPC UA Browser User Manual“, S. 28.

[35] „JavaScript basics - Learn web development | MDN“. https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics (zugegriffen Aug. 25, 2021).

[36] „Stack Overflow Developer Survey 2021“, *Stack Overflow*. https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021 (zugegriffen Aug. 25, 2021).

[37] „ECMAScript 6 compatibility table“. https://kangax.github.io/compat-table/es6/ (zugegriffen Sep. 11, 2021).

[38] „Members“, *OPC Foundation*. https://opcfoundation.org/members/ (zugegriffen Aug. 25, 2021).

[39] „NodeOPCUA! the OPCUA sdk for node.js“. https://node-opcua.github.io/ (zugegriffen Aug. 25, 2021).

[40] „Companies using Express“. https://expressjs.com/en/resources/companies-using-express.html (zugegriffen Aug. 25, 2021).

[41] Andrei Kashcha, „npm rank“, *Gist*. https://gist.github.com/anvaka/8e8fa57c7ee1350e3491 (zugegriffen Aug. 25, 2021).

[42] „Integrierte Entwicklungsumgebung“, *Wikipedia*. Mai 14, 2021. Zugegriffen: Okt. 24, 2021. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=Integrierte_Entwicklungsumgebung&oldid=211938103

[43] „State of JS 2020: Other Tools“. https://2020.stateofjs.com/en-US/other-tools/ (zugegriffen Okt. 24, 2021).

[44] „SOAP“, *Wikipedia*. Mai 18, 2021. Zugegriffen: Aug. 27, 2021. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=SOAP&oldid=212107412

[45] „How REST replaced SOAP on the Web: What it means to you“, *InfoQ*. https://www.infoq.com/articles/rest-soap/ (zugegriffen Aug. 27, 2021).

[46] „Google Trends (SOAP, REST)“, *Google Trends*. https://trends.google.com/trends/explore?date=2004-01-01%202014-12-31&geo=DE&q=rest%20api,SOAP%20api (zugegriffen Aug. 27, 2021).

[47] „GraphQL“, *Wikipedia*. Juli 13, 2021. Zugegriffen: Aug. 27, 2021. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=GraphQL&oldid=1033378613

[48] „About gRPC“, *gRPC*. https://grpc.io/about/ (zugegriffen Aug. 27, 2021).

[49] „CRUD“, *Wikipedia*. Apr. 13, 2020. Zugegriffen: Aug. 29, 2021. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=CRUD&oldid=198806386

[50] „Hypertext Transfer Protocol (HTTP) Status Code Registry“. https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml (zugegriffen Aug. 30, 2021).

[51] „JSON:API — Examples“. https://jsonapi.org/examples/ (zugegriffen Aug. 30, 2021).

[52] „Resource Caching - JSONAPI::Resources“. https://jsonapi-resources.com/v0.9/guide/resource_caching.html (zugegriffen Aug. 31, 2021).

[53] „Protocol Buffers“, *Google Developers*. https://developers.google.com/protocol-buffers?hl=de (zugegriffen Aug. 31, 2021).

[54] „Protocol Buffers“, *Wikipedia*. Juli 09, 2019. Zugegriffen: Aug. 31, 2021. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=Protocol_Buffers&oldid=190284588

[55] „Introduction to gRPC“, *gRPC*. https://grpc.io/docs/what-is-grpc/introduction/ (zugegriffen Aug. 31, 2021).

[56] „Beating JSON performance with Protobuf“, *Auth0 - Blog*. https://auth0.com/blog/beating-json-performance-with-protobuf/ (zugegriffen Aug. 31, 2021).

[57] „Comparing sizes of protobuf vs json | Random codewalks“. https://nilsmagnus.github.io/post/proto-json-sizes/ (zugegriffen Aug. 31, 2021).

[58] „HTTP/2 vs. HTTP/1.1: How do they affect web performance?“, *Cloudflare*. https://www.cloudflare.com/de-de/learning/performance/http2-vs-http1.1/ (zugegriffen Aug. 31, 2021).

[59] „Error handling“, *gRPC*. https://grpc.io/docs/guides/error/ (zugegriffen Sep. 01, 2021).

[60] A. Galloni, R. Marx, und M. Bishop, „The 2020 Web Almanac: HTTP/2“, HTTP Archive, Dez. 2020. Zugegriffen: Aug. 31, 2021. [Online]. Verfügbar unter: https://almanac.httparchive.org/en/2020/http2

[61] „Introduction to GraphQL | GraphQL“. https://graphql.org/learn/ (zugegriffen Sep. 01, 2021).

[62] „Subscriptions in GraphQL and Relay | GraphQL“. https://graphql.org/blog/subscriptions-in-graphql-and-relay/ (zugegriffen Sep. 06, 2021).

[63] „Google Trends“, *Wikipedia*. Nov. 19, 2020. Zugegriffen: Sep. 15, 2021. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=Google_Trends&oldid=205731477

[64] „Google Trends (REST, JSON:API, gRPC, GraphQL)“, *Google Trends*, Sep. 15, 2021. https://trends.google.com/trends/explore?date=2016-01-01%202021-09-15&q=rest%20api,json:api,graphql,grpc (zugegriffen Sep. 15, 2021).

[65] „SmartBear Software“, *Wikipedia*. Aug. 13, 2021. Zugegriffen: Sep. 15, 2021. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=SmartBear_Software&oldid=1038644186

[66] „State of API 2020 Report | SmartBear“. https://smartbear.com/resources/ebooks/the-state-of-api-2020-report/ (zugegriffen Aug. 25, 2021).

[67] SmartBear, „State of API 2019 Report | SmartBear“. 2019.

[68] RapidAPI, „RapidAPI Developer Survey and Insights 2019 - 2020“.

[69] „Postman API Platform | Sign Up for Free“, *Postman*. https://www.postman.com/ (zugegriffen Sep. 20, 2021).

[70] „2020 State of the API Report | API Technologies“, *Postman*. https://www.postman.com/state-of-api/api-technologies/ (zugegriffen Aug. 26, 2021).

[71] Ferdinand Malcher, „Untersuchung der Performance einer Web-API mit GraphQL und REST“, HTWK Leipzig, Leipzig, 2018.

[72] „Polling (computer science)“, *Wikipedia*. Nov. 07, 2020. Zugegriffen: Sep. 21, 2021. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=Polling_(computer_science)&oldid=987548890

[73] „Push technology“, *Wikipedia*. Aug. 09, 2021. Zugegriffen: Sep. 21, 2021. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=Push_technology&oldid=1037882618

[74] „Duplex (Nachrichtentechnik)“, *Wikipedia*. Juli 21, 2020. Zugegriffen: Sep. 22, 2021. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=Duplex_(Nachrichtentechnik)&oldid=202094109

[75] „WebSocket“, *Wikipedia*. Sep. 15, 2021. Zugegriffen: Sep. 22, 2021. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=WebSocket&oldid=1044480218

[76] „WebSockets vs Long Polling“, *Ably Blog: Data in Motion*. https://ghost.ably.com/blog/websockets-vs-long-polling/ (zugegriffen Sep. 22, 2021).

[77] „HTML Standard“. https://html.spec.whatwg.org/multipage/server-sent-events.html#the-eventsource-interface (zugegriffen Sep. 22, 2021).

[78] „Server-Sent Events (SSE): A Conceptual Deep Dive“, *Ably Realtime*. https://ably.com/topic/server-sent-events (zugegriffen Sep. 22, 2021).

[79] „Using server-sent events - Web APIs | MDN“. https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events (zugegriffen Sep. 22, 2021).

[80] „Push API - Web APIs | MDN“. https://developer.mozilla.org/de/docs/Web/API/Push_API (zugegriffen Sep. 22, 2021).

[81] „How Push Works | Web Fundamentals“, *Google Developers*. https://developers.google.com/web/fundamentals/push-notifications/how-push-works?hl=de (zugegriffen Sep. 22, 2021).

[82] „Push-Model“, *Wikipedia*. Sep. 30, 2020. Zugegriffen: Sep. 22, 2021. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=Push-Model&oldid=204127510

[83] „Google Trends (WebSockets, SSE, Push, Polling)“, *Google Trends*. https://trends.google.com/trends/explore?date=2012-01-01%202021-09-22&geo=DE&q=%2Fm%2F09rsvzj,%2Fm%2F0bbzryw,Push%20api,%2Fm%2F0c6p9v (zugegriffen Sep. 22, 2021).

[84] Paul Murley, Joshua Mason, Zane Ma, Michael Bailey, und Amin Kharraz, „WebSocket Adoption and the Landscape of the Real-Time Web“, New York, 2021. Zugegriffen: Sep. 22, 2021. [Online]. Verfügbar unter: https://doi.org/10.1145/3442381.3450063

[85] „A research-oriented top sites ranking hardened against manipulation - Tranco“. https://tranco-list.eu/ (zugegriffen Sep. 22, 2021).

[86] „Stack Overflow Trends (JSON)“. https://insights.stackoverflow.com/trends?tags=xml%2Cjson%2Cprotocol-buffers%2Ccsv%2Cyaml (zugegriffen Okt. 24, 2021).

[87] „IANA — IANA-managed Reserved Domains“. https://www.iana.org/domains/reserved (zugegriffen Sep. 26, 2021).

[88] „draft-kelly-json-hal-06“ Zugegriffen: Sep. 29, 2021. [Online]. Verfügbar unter: https://datatracker.ietf.org/doc/html/draft-kelly-json-hal-06

[89] A. Melnikov und I. Fette, „rfc6455 - The WebSocket Protocol“, Dez. 2011 Zugegriffen: Okt. 02, 2021. [Online]. Verfügbar unter: https://datatracker.ietf.org/doc/html/rfc6455

[90] „Express Tutorial Part 4: Routes and controllers - Learn web development | MDN“. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes (zugegriffen Okt. 15, 2021).

[91] „debug“, *npm*. https://www.npmjs.com/package/debug (zugegriffen Okt. 16, 2021).

[92] „morgan“, *npm*. https://www.npmjs.com/package/morgan (zugegriffen Okt. 16, 2021).

[93] „OPC 10000-3 - UA Specification Part 3 - Address Space Model 1.04“, *OPC Foundation*. https://opcfoundation.org/members/ (zugegriffen Aug. 05, 2021).

[94] „express-ws“, *npm*. https://www.npmjs.com/package/express-ws (zugegriffen Okt. 16, 2021).

[95] „VariantOptions | NodeOPCUA reference documentation - public API“. https://node-opcua.github.io/api_doc/2.32.0/interfaces/node_opcua.variantoptions.html (zugegriffen Okt. 17, 2021).

[96] „Date - JavaScript | MDN“. https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Date (zugegriffen Okt. 17, 2021).

[97] „What is Swagger“. https://swagger.io/docs/specification/2-0/what-is-swagger/ (zugegriffen Okt. 27, 2021).

[98] OPCFoundation, „OPC 10000-12 Unified Architecture Part 12 Discovery and Global Services“, *OPC UA Online Reference*. https://reference.opcfoundation.org/v104/GDS/docs/ (zugegriffen Okt. 27, 2021).

[99] *node-opcua*. node-opcua, 2021. Zugegriffen: Sep. 25, 2021. [Online]. Verfügbar unter: https://github.com/node-opcua/node-opcua/blob/3ec6296e8fd4df0e05b9d887f0b7dd20113a9ce1/README.md

[100] „MQTT - The Standard for IoT Messaging“. https://mqtt.org/ (zugegriffen Okt. 27, 2021).

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel als die angegebenen verwendet habe.

Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

_____

Ort, Datum, Unterschrift