Zentralinstitut für Engineering, Elektronik und
Analytik (ZEA) · Systeme der Elektronik (ZEA-2)

Konzeptionierung einer universellen Datenschnittstelle für die Ansteuerung von Labormessgeräten mittels SCPI

Daniel Keßel

Jül-4438

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum

Zentralinstitut für Engineering, Elektronik und
Analytik (ZEA) · Systeme der Elektronik (ZEA-2)

Konzeptionierung einer universellen Datenschnittstelle für die Ansteuerung von Labormessgeräten mittels SCPI

Daniel Keßel

Berichte des Forschungszentrums Jülich
Jül-4438 · ISSN 0944-2952
Zentralinstitut für Engineering,
Elektronik und Analytik (ZEA)
Systeme der Elektronik (ZEA-2)
82 (Bachelor FH Aachen, Campus Jülich, 2022)

Vollständig frei verfügbar über das Publikations-
portal des Forschungszentrums Jülich (JuSER)
unter www.fz-juelich.de/zb/openaccess

Forschungszentrum Jülich GmbH · 52425 Jülich
Zentralbibliothek, Verlag
Tel.: 02461 61-5220 · Fax: 02461 61-6103
zb-publikation@fz-juelich.de
www.fz-juelich.de/zb

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.
Ich verpflichte mich, ein Exemplar der Bachelorarbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Ort, Datum	Daniel Keßel

Kurzfassung

Im Rahmen dieser Arbeit wird ein Konzept für eine universale Datenschnittstelle erstellt. Diese Schnittstelle soll für die Kommunikation mit *SCPI*-Geräten in *Messskripten* genutzt werden. Sie erlaubt das Definieren von geräteübergreifenden Funktionen, welche von der Datenschnittstelle in die, für das Geräte vorgesehenen, *SCPI*-Befehle überführt werden. Das Ziel dieser Datenschnittstelle ist es, die Entwicklung neuer *Messskripte* zu vereinfachen und den Austausch von Messgeräten innerhalb eines *Messskriptes* mit möglichst wenig Änderungen zu ermöglichen.

Um dies zu erreichen, werden für jedes Gerätemodell die Funktionen definiert, welche von dem Gerät unterstützt werden. Dafür werden den Funktionen die nötigen *SCPI*-Befehle des Gerätes hinterlegt. Es ist dabei ebenfalls möglich, Parameter für die Funktionen zu definieren. Dafür werden die möglichen Datentypen und optional auch mögliche Werte oder Wertebereiche der Parameter angegeben. In den *Messskripten* werden daraufhin vor dem Absenden eines Befehls diese Restriktionen an die Parameter kontrolliert, um einen fehlerhaften *SCPI*-Befehl frühzeitig erkennen zu können.
Mögliche Antworten der *SCPI*-Befehle werden ebenfalls hinterlegt. Diese werden durch die Datenschnittstelle eingelesen, in die einzelnen Werte zerlegt und anhand der Formate der *SCPI*-Datentypen in die entsprechenden Datentypen der Programmiersprache umgewandelt. Die ausgewerteten Daten werden im *Messskript* wieder zur Verfügung gestellt, um weiterverwendet oder gespeichert zu werden.

Es können außerdem Geräteklassen definiert werden, welche einen bestimmten Befehlssatz voraussetzen. Wenn ein Gerät eine Geräteklasse implementiert, müssen deren vorgegebene Funktionen unterstützt werden. Somit können Geräte identischer Geräteklassen ohne Änderungen am Ablauf des *Messskriptes* ausgetauscht werden.

Für die Speicherung der Daten wird die dokumentenorientierten Datenbank *MongoDB* genutzt. Für diese werden fünf Kollektionen entworfen, welche die nötigen Geräteinformationen enthalten. Die in den Kollektionen enthaltenen Dokumente werden zusätzlich durch sogenannte *Schema-Validatoren* bezüglich ihr Formate kontrolliert. Für das Auslesen der Geräteinformationen wurden mehrere *Pipelines* entwickelt, welche die gespeicherten Dokumente zusammenfügen und verarbeiten. Diese Dokumente werden von der Datenschnittstelle genutzt, um die Befehle im *Messskript* auszuführen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Dokumente und Programmcode

1. Einleitung

In diesem Kapitel wird zunächst das *Zentralinstitut für Engineering, Elektronik und Analytik* der Forschungszentrum Jülich GmbH vorgestellt. Daraufhin folgt in Unterkapitel 1.2 die Motivation für diese Arbeit.

1.1. Zentralinstitut für Engineering, Elektronik und Analytik

Das Zentralinstitut für Engineering, Elektronik und Analytik (ZEA) des Forschungszentrums Jülich entwickelt Prozesse, Geräte, Experimente, Analyseverfahren, Mess-, Analyse- und Regelungsanlagen, Detektorsysteme, sowie computergestützte Werkzeuge und bildgebende Verfahren für die Spitzenforschung. Diese Entwicklungen werden in Kooperation mit anderen Instituten des Forschungszentrums, oder mit externen Partnern erarbeitet.[1]

Das ZEA setzt sich dabei aus drei Bereichen zusammen.

1. Engineering und Technologie (ZEA-1)
2. Systeme der Elektronik (ZEA-2)
3. Analytik (ZEA-3)

Diese Arbeit wurde am ZEA-2 erstellt. Deshalb wird dieser Bereich im Folgenden kurz vorgestellt.

Systeme der Elektronik (ZEA-2)

Das ZEA-2 fokussiert sich auf die Entwicklung von integrierten Systemen, Hardware- und Softwaresystemen, sowie *a*uf System-on-a-Chip (SoC). Der Schwerpunkt liegt dabei in der Entwicklung von komplexen elektronischen und informationstechnischen

Systemen für die Forschung. Diese reichen von der Erfassung der physikalischen Ereignisse bis zur Informationsextraktion. Die anwendungsübergreifenden Konzepte basieren auf existierenden sowie selbst entwickelten Technologien.[2]

1.2. Motivation

In diesem Unterkapitel wird zunächst auf das aktuelle Vorgehen bei Labormessungen im ZEA-2 eingegangen. Daraufhin werden in Abschnitt 1.2.2 Möglichkeiten für die Optimierung des aktuellen Vorgehens erläutert, welche schließlich in Abschnitt 1.2.3 zu der Motivation für die Erstellung der universellen Datenschnittstelle führen.

1.2.1. Labormessungen im ZEA-2

Das ZEA-2 führt unterschiedlichste Labormessungen durch. Ein Beispiel für diese ist die Verifikation der selbst entwickelten integrierten Systeme. Dabei können sowohl die Funktionalität, als auch die gesetzten Anforderungen an die Geräte, wie die maximale Stromaufnahme, kontrolliert werden.
Für die Labormessungen werden unterschiedlichste Geräte genutzt. Zu diesen gehören Oszilloskope, Multimeter, Wärme- und Kälteschränke und viele mehr. Diese Geräte unterstützen sogenannte *SCPI*-Befehle. Dies ist ein Befehlsformat für programmierbare Instrumente, auf welches in Kapitel 2 auf Seite 5 genauer eingegangen wird.
Die Messungen werden durch sogenannte *Messskripte* umgesetzt. Ein *Messskript* besteht aus einer Abfolge dieser *SCPI*-Befehle und übernimmt durch diese die Steuerung der Geräte. Diese *Messskripte* sind häufig in *MATLAB*[1] erstellt und nutzen für die Kommunikation mit dem Gerät die *Virtual instrument software architecture*[2], auch `VISA` genannt.

Wenn innerhalb eines Messvorgangs ein Gerät getauscht werden muss, müssen alle gerätespezifischen Befehle in den *Messskripten* abgeändert werden, um mit dem neuen Messgerät kompatibel zu sein. Dies führt zu einem hohen Aufwand, um einzelne Geräte auszutauschen. Ein Grund für den Tausch zweier Geräte wäre zum Beispiel, dass eines der Geräte defekt ist, oder weil ein Gerät mit anderen Funktionen, wie einer höheren Auflösung, benötigt wird.
Außerdem führt dies zu einem hohen zeitlichen Aufwand in der Entwicklung der *Messskripte*, da für jedes Gerät die entsprechenden Funktionen und *SCPI*-Befehle aus ihren Dokumentationen gesucht und in die *Messskripte* integriert werden müssen.

[1] MATLAB ist eine von MathWorks® entwickelte Programmiersprache. Weitere Informationen könnten unter folgendem Link gefunden werden: `https://de.mathworks.com/products/matlab.html`

[2] Die *Virtual instrument software architecture* ist eine von der *IVI Foundation* definierte Software-

1.2.2. Möglichkeiten einer Optimierung des aktuellen Vorgehens

Beim aktuellen Vorgehen werden die einzelnen Befehle der Geräte fest in die *Messskripte* integriert. Dadurch ist es nicht ohne weiteres möglich, ein Messgerät gegen ein anderes Modell auszutauschen, ohne zusätzlich die *SCPI*-Befehle in den Skripten anzupassen. Dies ist besonders der Fall, wenn die Geräte von einem anderen Hersteller sind. Dies liegt daran, dass die Geräte unterschiedliche *SCPI*-Befehle für die gleichen Funktionen nutzen können.
Diese Anpassung muss für jedes ausgetauschte Gerät vorgenommen werden und ist somit zeitaufwändig. Außerdem kommt es häufig zur Wiederholung der Arbeit, da die Anpassungen für ein anderes Gerät nur in dem derzeit verwendeten *Messskript* vorgenommen wurden, jedoch in anderen *Messskripten* nicht. Es ist zudem auch ein größerer Aufwand für eine bestimmte Funktion die entsprechenden *SCPI*-Befehle aus den Dokumentationen der Geräte zu suchen und in die *Messskripte* zu integrieren.
Ein weiteres Problem ist, dass Geräte mehrere Befehle für das vornehmen einer Einstellung unterstützen können. Dadurch ist es abhängig vom Labormitarbeiter, wie diese Einstellungen getroffen werden und es existiert bisher kein einheitliches Vorgehen.

1.2.3. Ziel einer universellen Datenschnittstelle für SCPI Befehle

Durch die Entwicklung einer universellen Datenschnittstelle sollen die zuvor in Abschnitt 1.2.2 genannten Verbesserungsmöglichkeiten erreicht werden.
Sie soll eine einheitliche Schnittstelle für die Programmierung (*API*) bereitstellen, welche allgemeine Anfragen, wie zum Beispiel das Setzen oder Auslesen einer Spannung, in die entsprechenden *SCPI*-Befehle überführen kann. Diese können daraufhin an das Gerät geschickt werden, unabhängig von dem genutzten Modell oder Hersteller.
Die Datenschnittstelle soll dabei so konzipiert sein, dass zukünftig auch weitere Befehlssätze hinzugefügt werden können. Zu diesen gehört zum Beispiel die Übertragung der Daten über *JTAG*, *SPI* oder I^2C.
Des Weiteren soll die Schnittstelle mehrere Programmiersprachen unterstützen können, sodass die Labormitarbeiter eine größere Auswahl an Programmiersprachen erhalten.
Schließlich soll die Mehrarbeit reduziert werden, welche aus dem Suchen der benötigten *SCPI*-Befehle entsteht. Dafür sollen die Geräte nur einmalig in die Datenschnittstelle hinzugefügt werden und anschließend über diese nutzbar sein.
Die Nutzung der Laborgeräte und die Erstellung von Messungen durch *Messskripte*

Bibliothek, unter anderem für die Ansteuerung von Messgeräten. Weitere Information können unter diesem Link gefunden werden: `https://ivifoundation.org/specifications/default.aspx`

soll somit schneller und einfacher möglich sein. Ein Austausch eines Geräts bei einer bereits existierenden Messumgebung wird dadurch ebenfalls vereinfacht.

2. Standard Commands for Programmable Instruments (SCPI)

Der *Standard-Commands-for-Programmable-Instruments (SCPI)* Standard wurde im Jahre 1992 vom *SCPI Consortium* entwickelt und 1999 erneuert. Er definiert die Syntax eines gemeinsamen Befehlssatzes für die Nutzung von programmierbaren Instrumenten. Seit 2002/2003 ist das *SCPI Consortium* Teil der *Interchangeable Virtual Instruments Foundation (IVI Foundation)*. [3, 4, 5]

SCPI basiert auf den *IEEE*-Standards *488.1* und *488.2*.
Der *IEEE-488.1* Standard wurde im Jahr 1975 entwickelt und 1987 aktualisiert. Er definiert den `General Purpose Interface Bus (GPIB)` als Hardware-Schnittstellen für (Mess-)Geräte.[6]
IEEE-488.2 definiert hingegen die Kommunikation zwischen einem Gerät und dem Kontrollsystem. Es wird außerdem definiert, wie ein Befehl an ein Gerät gesendet werden kann und wie die Antwort des Gerätes übertragen wird. Zusätzlich zum Aufbau dieser Nachrichten wurde ein Befehlssatz von regulären Befehlen definiert, welchen die Geräte für eine Kompatibilität mit dem *IEEE-488.2*-Standard unterstützen müssen. Die Entwicklung der gerätespezifischen Befehle wurde allerdings den Herstellern der Geräte überlassen. *IEEE-488.2* wurde im Jahre 1987 entwickelt und 1992 aktualisiert.[7]

Da der *IEEE-488.2* Standard den Herstellern die Entwicklung der Befehle überließ, wurden für Geräte mit ähnlichen Funktionen, aber unterschiedlichen Herstellern, häufig stark unterschiedliche Befehlssätze entwickelt. Aus diesem Grund wurde der *SCPI* Standard erstellt. Dessen Ziel ist die Erstellung einer Umgebung, in welcher der Austausch von Messgeräten vereinfacht werden soll.
Aufbauend auf der in *IEEE-488.2* definierten Übertragung definiert der *SCPI* Standard deshalb ein Schema für die Erstellung der gerätespezifischen Befehle. Es wird jedoch nicht vorgeschrieben, welche Verbindung zwischen dem Kontrollsystem und dem Gerät genutzt werden muss. Viele Messgeräte unterstützen daher auch die Übertragung von *SCPI*-Befehlen über Anschlüsse wie *USB* oder *Ethernet* mittels *TCP/IP*.

Der *SCPI*-Standard besteht aus vier Bänden. In dieser Arbeit wird nur auf den ersten Band „*Volume 1: Syntax and Style*“ genauer eingegangen. Die Bände „*Volume 2:*

Command Reference“, „*Volume 3: Data Interchange Format*“ und „*Volume 4: Instrument Classes*“ werden nicht weiter betrachtet.[8]

In den folgenden Unterkapiteln sollen die Grundlagen für die Nutzung von *SCPI*-Befehlen erläutert werden. Dafür wird im folgenden Unterkapitel näher auf die Syntax der Befehle eingegangen.

2.1. Befehlsformat

Ein *SCPI*-Befehl wird durch eine Zeichenkette aus *ASCII* Werten (im Folgenden auch *String* genannt) realisiert. Jeder der Befehle beginnt mit einem *Program Header*, welcher den auszuführenden Befehl identifiziert.
Es gibt zwei Arten von Befehlen. Die *Commands* welche Einstellungen in dem Gerät ändern und die *Queries* welche Daten vom Gerät abfragen und ausgeben.
Eine *Query* wird durch das Anhängen eines „`?`“ an den *Programm Header* realisiert.

Anschließend an den *Program Header* können die Parameter für den entsprechenden Befehl angefügt werden. Falls mehrere Parameter angegeben werden, werden diese durch ein „`,`“ voneinander getrennt.
Schließlich können auch mehrere Befehle gleichzeitig übertragen werden. Dafür werden die Befehle voneinander durch ein „`;`“ getrennt.
Um den Befehl abzuschließen wird ein *NewLine* (*ASCII* Wert 10) gesendet und die Kommunikation mit dem Gerät wird beendet.

Die *SCPI*-Befehle werden außerdem in Standardbefehle (*Common-Commands*) und gerätespezifische Befehle unterschieden. Auf diese wird in den folgen Unterkapiteln genauer eingegangen. [7, 8]

2.2. Standardbefehle

Der *SCPI* Standard definiert eine Liste von Standardbefehlen, welche von allen *SCPI*-kompatiblen Geräten unterstützt werden müssen. Diese Befehle sind dem *IEEE-488.2* Standard entnommen. Jeder dieser Befehle beginnt mit ein „`*`“, gefolgt von dem *Program Header*.
Der *Programm Header* identifiziert den auszuführenden Befehl eindeutig durch ein sogenanntes *Mnemonic*. Ein *Mnemonic* ist ein einzelnes Wort, welches dem Befehl

oder einer Funktion zugeschrieben ist. *Mnemonics* werden nach der in Abbildung 2.1 dargestellten Grammatik definiert.

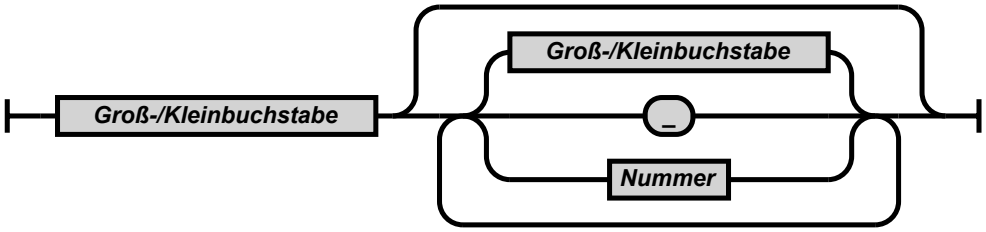

Abbildung 2.1.: Mnemonic Grammatik [7]

Ein Beispiel für einen *Common Command* sind der *Command* „`*RST`“, welcher das Gerät und einen Großteil seiner Einstellungen zurücksetzt, oder die *Query* „`*IDN?`“, welcher unter anderem den Hersteller und das Modell des Gerätes ausgibt.

`RST` und `IDN` sind die *Mnemonics* der *Common Commands*. Die Groß- und Kleinschreibung der *Mnemonics* wird ignoriert. Alle *Common Commands* folgen dem in Abbildung 2.2 gezeigten Aufbau.

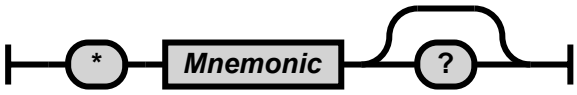

Abbildung 2.2.: Common Commands Grammatik [7]

Eine vollständige Liste der *Common-Commands* kann im Abschnitt 4.1.1 des *SCPI*-Standards gefunden werden. [7, 8]

2.3. Subsysteme

Die *SCPI*-Befehle, welche nicht den *Common-Commands* angehören, werden in sogenannte *Subsyteme* unterteilt. Jedes gehört dabei einer übergeordneten Funktion an, welche vom Gerät umgesetzt wird.
Die Befehle in einem *Subsystem* sind hierarchisch wie ein Baum definiert, sodass der gleiche Befehl in unterschiedlichen *Subsystemen* genutzt werden kann.

Im Gegensatz zu den *Common-Commands* beginnen die Befehle der *Subsysteme* mit einem „`:`“ gefolgt von dem *Mnemonic* des *Subsystems*. Die einzelnen Ebenen, auch

Nodes genannt, werden durch weitere „**:**“ getrennt. In jeder Ebene muss erneut ein *Mnemonic* angegeben werden. Zusammen ergeben der Name des *Subsystems* und dessen Ebenen den *Instrument-Control Header*, welcher den Befehl angibt.

Diese *Mnemonics* der *Subsysteme* gibt es sowohl in einer Kurz- als auch einer Langform. Bei der Angabe eines SCPI-Befehls wird üblicherweise der Teil der Kurzform großgeschrieben, während die Langform kleingeschrieben wird.

Ein Beispiel für einen *Instrument-Control Header* ist „`:DISPlay:WINDow:TEXT:DATA`“. Dieser Befehl gehört dem *Subsytem* „`Display`“ bzw. „Disp“ an. Der gleiche Befehl kann auch in seiner Kurzform als „`:DISP:WIND:TEXT:DATA`“ angegeben werden.

Die einzelnen Ebenen bzw. *Nodes* können selbst Befehle sein, so dass z.B. „`:DISP:WIND`“ bereits ein anderer Befehl sein kann. Die Befehle befinden sich somit nicht immer am Ende des erstellen „Baums“.

Auf den *Instrument-Control Header* können dann, wie bei den *Common-Commands*, die Parameter für den Befehl folgen.

Ein vollständiger SCPI-Befehl kann wie folgt aussehen:

```
:DISP:WIND:TEXT:DATA "Ausgabe auf Geraetedisplay"
```

Dieser Befehl könnte das Gerät anweisen, auf seinem Display den Text „Ausgabe auf Geraetedisplay“ anzuzeigen.

Die entsprechende Abfrage oder *Query* des Textes ist:

```
:DISP:WIND:TEXT:DATA?
```

Mit dieser Abfrage wird der aktuell angezeigte Text vom Gerät übergeben. Es müssen nicht immer ein zugehöriger *Command* und eine entsprechende *Query* existieren.

In einem Messgerät können *Subsysteme* oder deren *Nodes* mehrfach vorhanden sein. Dies ist zum Beispiel bei mehreren Displays möglich. In diesem Fall kann das entsprechende Subsystem durch die Angabe eines numerischen Suffixes angegeben werden. Ein Beispiel für dieses Suffix ist der Befehl:

```
:DISPlay3:WINDow:STATe?
```

Dieser fragt für das dritte Display den aktuellen Status ab.
Die Angabe eines Suffixes ist optional. Wenn kein Suffix angegeben wurde, wird der Befehl für das erste Modul ausgeführt, als wäre eine „1“ als Suffix angegeben.

Alle *Instrument-Control Header* der *Subsystem*-Befehle folgen der in Abbildung 2.3 gezeigten Grammatik. [7, 8]

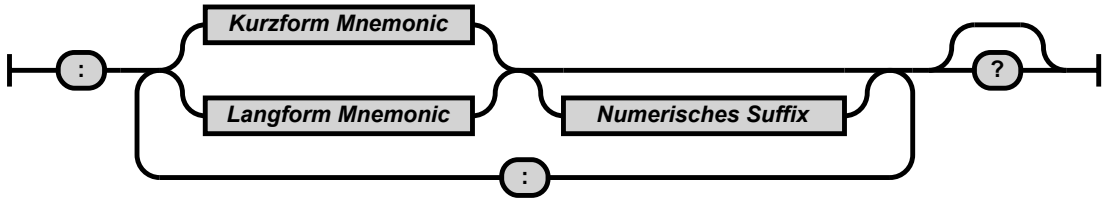

Abbildung 2.3.: Subsystem Grammatik [7]

2.4. Datentypen

Der *SCPI*-Standard definiert die unterschiedlichen Datentypen, welche als Parameter und Rückgabewerte genutzt werden können. Die meisten Datentypen sind dem *IEEE-488.2* Standard Sektion 7.7 entnommen, können aber durch den *SCPI*-Standard weiter eingeschränkt sein. [7, 8]

Auf die Datentypen wird im Folgenden kurz eingegangen.
Dabei werden diese nur vereinfacht dargestellt und entsprechen nicht genau dem *SCPI*-Standard. Die meisten Datentypen unterscheiden sich in ihrem Format, wenn sie als Rückgabewert vom Messgerät gesendet werden. In diesen Fällen werden die Formate der Datentypen weiter eingeschränkt. Auf die Unterschiede zwischen den erlaubten Eingabeformaten und den vorgeschriebenen Ausgabeformaten wird ebenfalls nicht näher eingegangen.
Eine vollständige Übersicht der Datentypen kann im *SCPI* Standard Abschnitt 7 und 8, sowie im *IEEE-488.2* Standard Abschnitt 7.7 und 8.7 gefunden werden.

Character

Befehle können *Character Program Data* als Parameter akzeptieren. Ein *Character Program Data* Parameter entspricht einem einzelnen *Mnemonic*.
Das Format eines Mnemonik wurde bereits in Abbildung 2.1 auf Seite 7 vorgestellt.

Einzelne *Mnemonics* werden genutzt, wenn die Angabe eines numerischen Datentypen nicht sinnvoll oder nicht möglich ist.
Beispiele für ein *Character Program Data* Parameter wären `MAXimum` und `MINimum`, welche den maximalen oder minimalen zulässigen Wert repräsentieren.[7, 8]

Numerisch

Unter numerischen Datentypen fasst der *SCPI*-Standard sowohl Ganzzahlen als auch Gleitkommazahlen zusammen. Die Gleitkommazahlen können ebenfalls in wissenschaftlicher Schreibweise angegeben werden.
Die Grafik 2.4 zeigt die Grammatik eines numerischen Datentypen. Zusätzlich ist in Grafik 2.5 die Definition der *Mantisse* dargestellt. In Grafik 2.6 ist die Definition des *Exponenten* zu sehen.

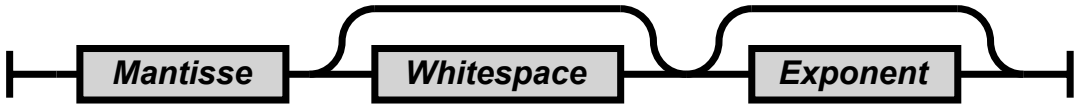

Abbildung 2.4.: Grammatik numerischer Datentypen [7]

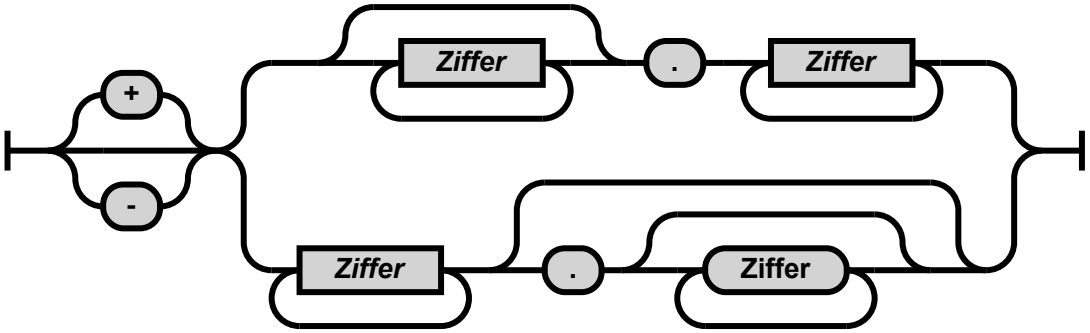

Abbildung 2.5.: Grammatik der Mantisse eines numerischen Datentypen [7]

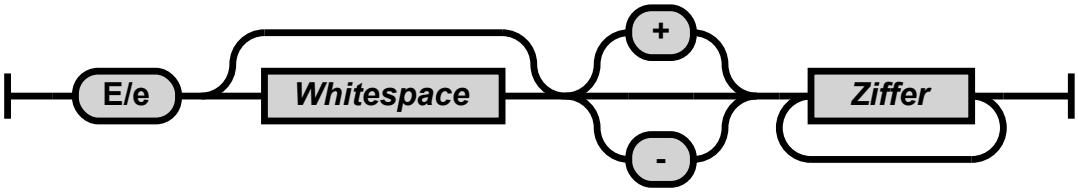

Abbildung 2.6.: Grammatik des Exponenten eines numerischen Datentypen [7]

Es werden zusätzlich spezielle Werte definiert. Dabei wird `9.9E37` als Unendlich (*Infinity* bzw. *INF*), `-9.9E37` als negativ Unendlich (*Negative Infinitiy* bzw. *NINF*) und `9.91E37` als „Keine Zahl“ (*Not a Number* bzw. *NaN*) definiert. Alle Werte, die über `9.9E37` und `-9.9E37` hinausgehen werden als Fehler angesehen.
Die Spezialwörter *INF*, *NINF* und *NAN* können anstelle von den Zahlenwerten ebenfalls in den Befehlen genutzt werden.[7, 8]

Nicht dezimale Ganzzahlen

Zusätzlich zu den numerischen Datentypen werden auch *hexadezimale*, *oktale* und *binäre* Werte genutzt. Sie werden durch ein „`#`“ eingeleitet, gefolgt von einem Zeichen, welches die genutzte Basis angibt. Für hexadezimale Werte ist dies ein „`H`“, für oktale ein „`Q`“ und für binäre ein „`B`“. Auf diese Basis folgt dann der eigentliche Wert der Zahl.

Nicht dezimale Zahlen sind durch die in Abbildung 2.7 dargestellte Grammatik definiert.[7]

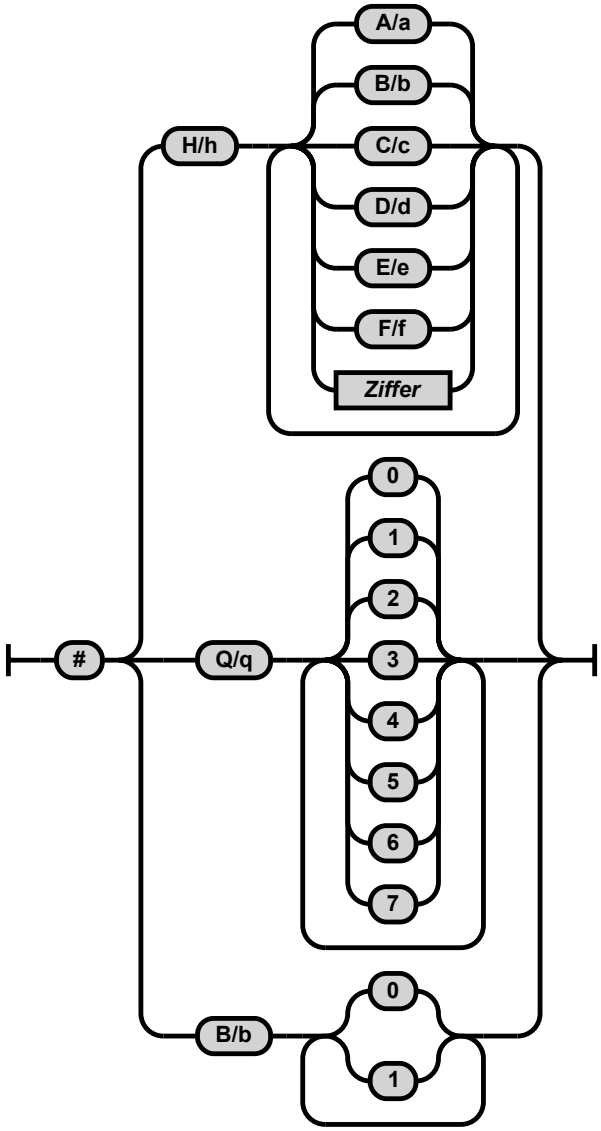

Abbildung 2.7.: Grammatik nicht dezimaler Datentypen [7]

Boolesche Werte

Der *SCPI*-Standard führt zusätzlich zu den Datentypen des *IEEE-488.2* Standards auch den booleschen Wert ein.

Der Wert `0` wird als *False* bzw. *Falsch* angesehen, ebenso wie das *Mnemonic* „`OFF`“. Als *True* bzw. *Wahr* werden alle Werte außer `0` und das *Mnemonic* „`ON`“ angesehen. Die eingegebenen numerischen Werte werden zu Ganzzahlen gerundet.[8]

String

Strings oder Zeichenketten werden durch Anführungszeichen eingeschlossen. Dabei sind einfache (') oder doppelte (") Anführungszeichen möglich. Zwischen den Anführungszeichen dürfen ausschließlich *ASCII* Werte stehen. Diese umfassen den Wertebereich von 0 bis 127 (bzw. 0x00 bis 0x7F *hexadezimal*).
Falls das einschließende Anführungszeichen innerhalb des *Strings* genutzt werden soll, muss dieses doppelt in dem *String* eingetragen werden. Dadurch wird ein einzelnes Anführungszeichen im *String* hinzugefügt.[7]

Die folgende Grammatik in Abbildung 2.8 beschreibt den Aufbau eines *Strings*.

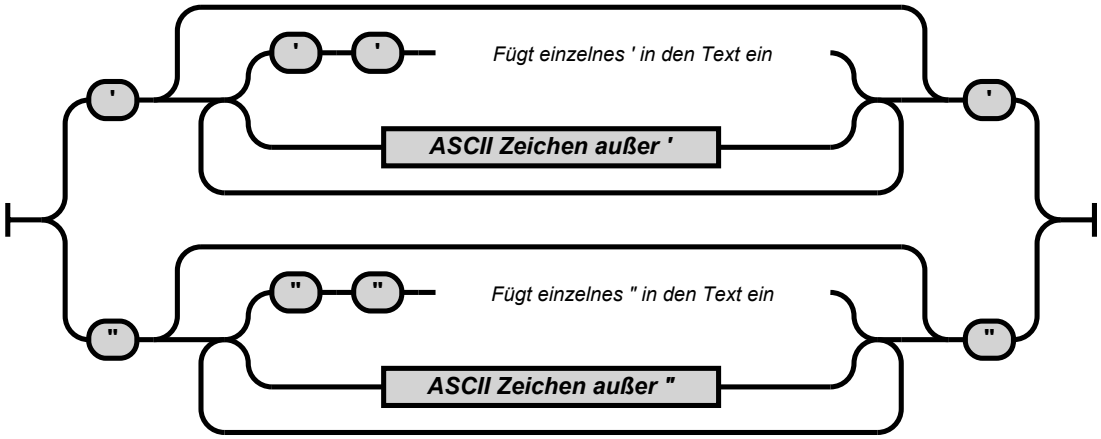

Abbildung 2.8.: Grammatik eines Strings [7]

Expression

Eine *Expression* ist ein durch runde Klammern eingeschlossener Ausdruck. Der *SCPI*-Standard definiert fünf unterschiedliche *Expressions*:

1. **Channel Lists**: Dient der Auswahl von elektrischen Anschlüssen am Gerät

2. **Numeric Expression**: Definiert einen numerischen Ausdruck, welcher vom Gerät berechnet wird

3. **Numeric Lists**: Dient zum kompakten Angeben von Nummern und Nummerbereichen

4. **Data Interchange Format Expression**: Format für den Austausch von Daten, definiert in Band 3 „*Data Interchange Format*“ des *SCPI*-Standards

5. **Instrument Specifier Expression**: Format zur Beschreibung der Gerätefunktionen, definiert in Band 4 „*Instrument Classes*“ des *SCPI*-Standards

Auf den Aufbau der definierten *Expressions* wird im Folgenden nicht weiter eingegangen. Diese Definitionen können in Band 1 Abschnitt 8 des *SPCI*-Standards gefunden werden.
Weitere *Expressions*, zusätzlich zu den definierten fünf, sind ebenfalls möglich.[7, 8]

Die generelle Definition einer *Expression* stellt die Grafik 2.9 dar.

Abbildung 2.9.: Grammatik einer Expression [7]

Arbitrary Program Data

Arbitrary Program Data besteht aus einem Block von binären Werten. Sie werden vor allem zum Übertragen von Werten außerhalb des *ASCII* Zeichensatzes genutzt.
Es wird in Blöcke mit definierter Länge und mit undefinierter Länge unterschieden.

Ein *Arbitrary Program Data* Block beginnt mit einem „`#`“, gefolgt von einer Ziffer. Wenn diese eine `0` ist, handelt es sich um einen Block mit unbestimmter Länge. Andernfalls gibt diese Ziffer an, wie viele der nachfolgenden Zeichen die Größe des Blockes beschreiben.
Ein Block aus fünf Datenbytes könnte zum Beispiel wie folgt eingeleitet werden:

#15<5 Datenbytes>	Eine Stelle gibt die Größe an, in diesem Fall „5“	(2.1)
#40005<5 Datenbytes>	Vier Stellen geben die Größe an, hier „0005“	(2.2)

Nach der Größenangabe folgt die angegebene Anzahl an Datenbytes.

Arbitrary Program Data Blöcke mit undefinierter Größe können beliebig viele Datenbytes enthalten. Der Block wird durch ein *NewLine* und dem Ende der Übertragung beendet. Es können daher keine weiteren Daten auf einen *Arbitrary Program Data* Block folgen, wenn dieser keine feste Größe besitzt.[7]

Die folgende Grammatik 2.10 beschreibt den Aufbau eines *Arbitrary Program Data* Blockes.

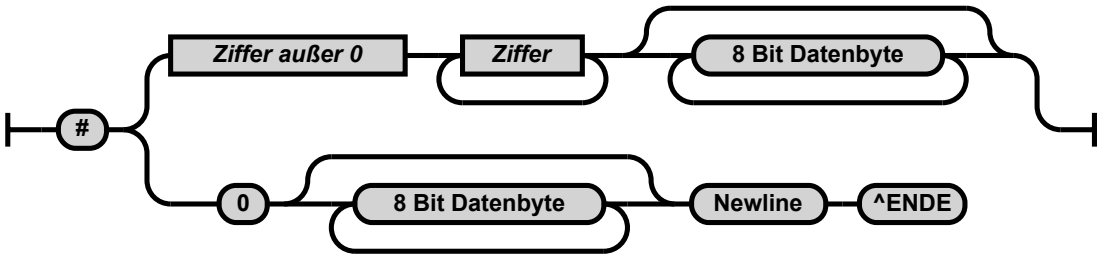

Abbildung 2.10.: Grammatik von *Arbitrary Program Data*[7]

Anmerkung: Das `^ENDE` meint die Beendigung der gesamten Übertragung.

Arbitrary ASCII

Arbitrary ASCII ist vergleichbar mit einem *Arbitrary Program Data* Block. Es erlaubt das Senden von einem beliebig langen Text im *ASCII* Format. Er wird durch ein *NewLine* und der Beendigung der gesamten Nachricht beendet. Daher können keine weiteren Daten nach einem *Arbitrary ASCII* Block folgen.[7]

Die Grammatik 2.11 beschreibt den Aufbau eines *Arbitrary ASCII* Blocks.

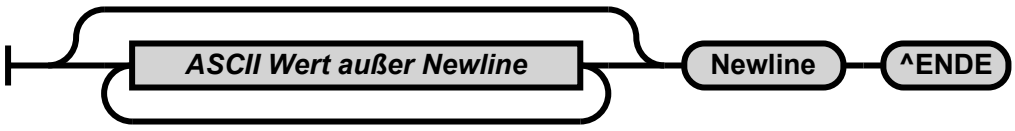

Abbildung 2.11.: Grammatik von *Arbitrary ASCII* [7]

Anmerkung: Das `^ENDE` meint die Beendigung der gesamten Übertragung.

Suffixe

Suffixe definieren eine physikalische Einheit. Sie können einzeln als Parameter angegeben werden, oder dezimalen numerischen Datentypen angehangen werden, wodurch

die Einheit der Daten angegeben wird. Dabei sind ausschließlich Einheiten erlaubt, welche für den gegebenen Parameter sinnvoll sind. Nicht dezimalen numerischen Daten darf kein Suffix angehangen werden.

Beispiele für Suffixe sind:

MA	$\Leftrightarrow$	Milliampere
MHZ	$\Leftrightarrow$	Megahertz
PS	$\Leftrightarrow$	Pikosekunden

und als Angabe der Einheit eines numerischen Werts:

5V	$\Leftrightarrow$	5 Volt
5000MV	$\Leftrightarrow$	5000 Millivolt
5E-6MAV	$\Leftrightarrow$	$0,000005$ Megavolt

Alle diese Werte werden als 5 Volt erkannt.

Auf die möglichen Suffixe, deren Multiplikatoren, sowie Grammatik wird nicht weiter eingegangen. Diese kann im *IEEE-488.2* Standard in den Abschnitten 7.7.3 und 8.7.13 gefunden werden.

3. Anforderungsanalyse

In diesem Kapitel soll zunächst eine Anforderungsanalyse erstellt werden. Dafür wird im ersten Unterkapitel generell auf das Projekt, für welches die Datenschnittstelle entworfen wird, näher eingegangen. In Unterkapitel 3.2 wird nachfolgend auf die benötigten Funktionen beschrieben und in Unterkapitel 3.3 werden schließlich die nötigen Daten für die Schnittstelle bestimmt.

3.1. Generell

Die Entwicklung der universellen Datenschnittstelle wird vor allem für die Forschung des ZEA-2 im Bereich des Quantencomputings entwickelt. Die zu erstellende Datenschnittstelle soll dabei nur ein Teil eines größeren, sogenannten *Laborsetups* werden, welches später für alle Messungen im Institut genutzt werden soll.
Ein weiterer Teil dieses *Laborsetups* ist die Verwaltung der Laborgeräte. In diesem Bereich wird von Klara Schnorrenberg zeitgleich eine Bachelorarbeit erstellt.[1]
Ihre Bachelorarbeit befasst sich mit der Konzeptionierung einer Datenmanagementsoftware, welche die Informationen für die einzelnen Laborgeräte verwaltet. Ziel dieser ist es einen besseren Überblick der Informationen zu geben, um zum Beispiel besser auf Kalibrierungstermine zu achten oder Messsetups erstellen zu können. In einem Messsetup können die Geräte, welche für eine Messung nötig sind, ausgewählt werden, um diese für den Zeitraum der Messung zu reservieren.
Damit innerhalb des *Laborsetups* auf dieselben Daten zugegriffen werden kann, ist ein gemeinsamer Datenspeicher nötig. Daher ist eine Planung der gemeinsamen Teile des *Laborsetups* in Zusammenarbeit mit Klara Schnorrenberg nötig. Auf diese Planung wird später in Kapitel 4 auf Seite 21 noch weiter eingegangen.

Abbildung 3.1 auf der nächsten Seite stellt die aktuelle Planung des *Laborsetups* dar.

[1] Bachelorarbeit von Klara Schnorrenberg mit dem Titel „Konzeptionierung einer Datenmanagementsoftware für die Verwaltung des Laborequipments“

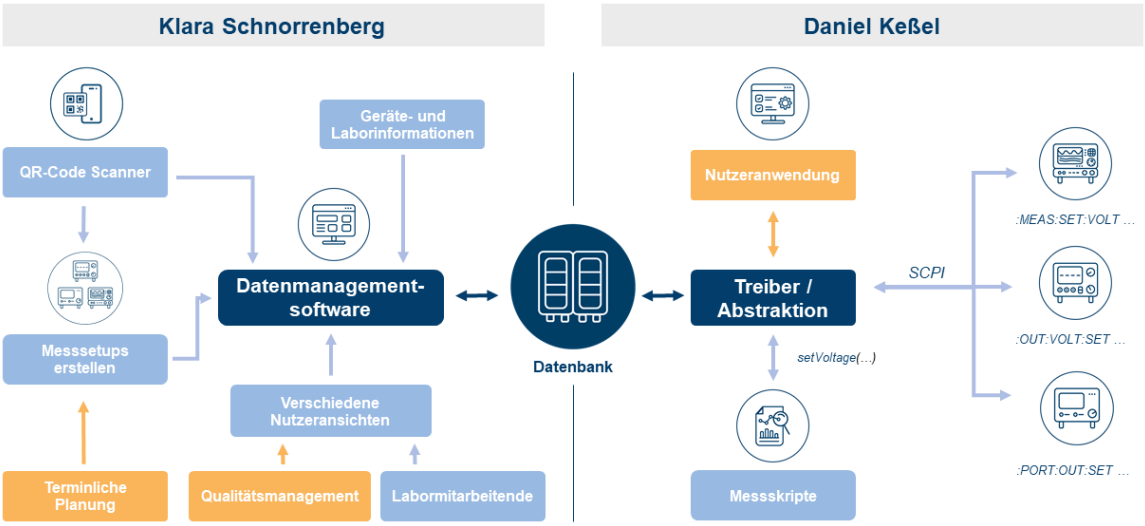

Abbildung 3.1.: Aktueller Projektplan des *Laborsetups*

In der linken Hälfte der Abbildung werden die Bestandteile der Datenmanagementsoftware von Klara Schnorrenberg dargestellt. Diese umfassen die Erstellung der Messsetups sowie die Verwaltung der Geräteinformationen. Die rechte Seite stellt die Teile der Datenschnittstelle und dieser Arbeit dar. Die in der Mitte gezeigte Datenbank steht hierbei für den gemeinsamen Datenspeicher.
In Orange eingefärbte Bestandteile sind bereits vorgesehen, aber nicht weiter in der jeweiligen Arbeit betrachtet. Die in der rechten Hälfte gezeigte *Nutzeranwendung*, für die Verwaltung der Datenschnittstelle, ist somit nicht Teil dieser Arbeit und wird nach der Realisierung umgesetzt.

3.2. Funktionalität

In diesem Abschnitt soll zunächst auf die grundlegenden Funktionalitäten eingegangen werden, welche für eine Nutzung der Datenschnittstelle in den *Messskripten* nötig sind.

Die Schnittstelle soll eine Abstraktion der Funktionalitäten von den *SCPI*-Befehlen ermöglichen. Um dies zu erreichen, muss die Datenschnittstelle das Definieren von Funktionen zulassen, welche von den entsprechenden *SCPI*-Befehlen umgesetzt werden. Ein Beispiel für eine solche Funktion ist „`setOutputStatus`“. Diese Funktion führt den hinterlegten *SCPI*-Befehl für das aktuelle Gerät aus. Dieser Befehl kann zum Beispiel „`:OUTPut:STATe`“ für das aktuelle Gerät sein. Der Aufruf einer Funktion, welche nicht von dem Gerät unterstützt wird, soll zu einem Fehler führen.
Für viele Funktionen sind Parameter nötig, welche mit in den *SCPI*-Befehl integriert

werden müssen. Diese sollen ebenfalls von der Schnittstelle verarbeitet werden. Dabei müssen auch die richtigen Reihenfolgen und Datentypen für die Parameter beachtet werden.
Ein Beispiel für einen vollständigen Funktionsaufruf ist „`setOutputStatus(true)`“, welcher durch die Datenschnittstelle in den *SCPI*-Befehl „`:OUTPut:STATe ON`“ umgewandelt werden kann. Diese Funktion könnte zum Beispiel den Ausgang einer Spannungsquelle einschalten.
Ebenso müssen die Rückgabedaten einer Funktion definiert werden können. Die Antwort des Gerätes, auf zum Beispiel eine *Query*, soll dann in die entsprechenden Datentypen übersetzt werden. Dies muss anhand der vorgegebenen Grammatiken der Datentypen geschehen, welche in Unterkapitel 2.4 auf Seite 9 bereits beschrieben wurden.

Außerdem muss die Schnittstelle für alle Labormitarbeiter in ihren *Messskripten* zugänglich sein und die darin enthaltenen Daten und Definitionen sollen aktuell sein.

3.3. Benötigte Daten

Um die Befehle für die Messgeräte umsetzen zu können, werden unterschiedliche Daten benötigt.
Zunächst werden die Modellnamen und Hersteller der Geräte benötigt, damit die Funktionen einem bestimmten Gerätemodell zugeschrieben werden können.
Die Funktionen der Geräte müssen darüber hinaus Namen erhalten können, sowie auch Funktionsparameter, welche für die Ausführung der Funktionen nötig sind.
Die *SCPI*-Befehle der einzelnen Geräte müssen ebenfalls, mit ihren möglichen Parametern, angegeben werden. Dabei ist auch relevant, in welcher Reihenfolge diese Parameter vorliegen, und welche dieser Parameter für eine Ausführung des Befehls nötig sind und welche optional sind.
Schließlich muss hinterlegt sein, welcher *SCPI*-Befehl für ein bestimmtes Gerätemodell genutzt werden soll, um eine der definierten Funktionen umzusetzen.

4. Konzeptionierung

In diesem Kapitel wird zunächst auf das grundlegende Konzept eingegangen, welches die grundlegenden Funktionen der Datenschnittstelle realisieren soll. Daraufhin werden in Unterkapitel 4.2 die Anforderungen an den Datenspeicher gestellt und unterschiedliche Datenbanktypen mit ihren Vor- und Nachteilen betrachtet. Anhand dieser Vor- und Nachteile wird schließlich in Abschnitt 4.2.3 ein Datenbankmanagementsystem (DBMS) ausgewählt, welches als gemeinsamer Datenspeicher genutzt werden soll.

4.1. Generelles Konzept

Um die Grundfunktionalitäten der Schnittstelle umsetzen zu können, müssen die Daten der Geräte gespeichert werden. Zunächst müssen die Gerätemodelle gespeichert werden. Dazu gehören der Modellname, der Name des Herstellers und die Funktionen, die das Gerät unterstützt. Dies beinhaltet die *SCPI*-Befehle, ihre Parameter mit möglichen Datentypen, die möglichen Antworten sowie deren Datentypen. Jeder Parameter und Rückgabewert erhält einen eindeutigen Namen, mit welchem er später identifiziert werden kann.
Es ist dabei zu beachten, welche Parameter für die korrekte Ausführung nötig sind. Zum Beispiel können Parameter in einem *SCPI*-Befehl optional sein. Falls aber einer dieser optionalen Parameter angegeben werden soll, müssen alle vorherigen Parameter ebenfalls angegeben werden, auch wenn diese eigentlich optional sind. Daher wird ebenfalls gespeichert, welche Kombinationen von Parametern möglich sind.
Manche Befehle benötigen für ihre korrekte Ausführung andere Befehle, welche zum Beispiel das Ausgabeformat eines Rückgabewertes bestimmen. Diese zusätzlichen Befehle sollen ebenfalls gespeichert und vor der Ausführung des eigentlichen Befehls vorangestellt werden.
Schließlich müssen ebenfalls die Verbindungsinformationen der einzelnen Geräte gespeichert werden. Diese sind jedoch Teil von Klara Schnorrenbergs Datenmanagementsoftware und daher in dieser Arbeit als gegeben angesehen.

Um die Funktionen definieren zu können, benötigen diese einen eindeutigen Namen, eine Liste an möglichen Parametern und mögliche Rückgabewerte. Die Parameter und

Rückgabewerte erhalten eine Reihenfolge und einen Namen, sodass sie in der späteren Nutzung in der richtigen Reihenfolge angegeben werden können, oder über ihren Namen festgelegt werden können.
Bei der Rückgabe wird die Reihenfolge der Rückgabewerte beachtet, wenn eine Rückgabe mehrerer Werte in der genutzten Programmiersprache möglich ist. Ansonsten sollen die Rückgabewerte über ihren Namen im *Messskript* aus der Rückgabe auslesbar sein.

Damit eine Funktion einem *SCPI*-Befehl zugewiesen werden kann, müssen diese für jedes Gerät eindeutig zueinander zugeordnet werden. Dabei ist zu beachten, dass die Definition eines *SCPI*-Befehls eventuell andere Namen für die Parameter nutzt, als die Definition der Funktion. Um dies zu umgehen, sollen die Namen der Funktionsparameter auf die der *SCPI*-Parameter abgebildet werden.
Es werden außerdem *Defaults* vorgesehen, da eine Funktion weniger Parameter zur Verfügung stellen könnte, als von dem *SCPI*-Befehl vorgesehen werden. In diesem Fall werden die übrigen Parameter durch einen bekannten *Defaultwert* festgelegt.

Während der Ausführung des *Messskriptes* sollen die Geräte durch ihren Modellnamen und Hersteller angegeben werden. Anhand dieser Daten werden daraufhin die Funktionen des Gerätes ausgelesen.
Wenn eine Funktion für ein Gerät ausgeführt werden soll, wird diese per Namen aufgerufen und die nötigen Parameter werden übergeben. Je nach Unterstützung durch die Programmiersprache soll dabei sowohl die Übergabe per Namen, als auch die Übergabe über die Reihenfolge der Parameter möglich sein.
Die Parameter werden daraufhin in die *SCPI*-Befehle integriert. Dabei werden sie gegebenenfalls in die entsprechenden Datentypen konvertiert, sofern dies möglich ist. Der somit erstellte *SCPI*-Befehl wird dann an das Gerät übertragen.
Falls eine Antwort auf den Befehl vorgesehen ist, wird diese eingelesen. Anhand der definierten Rückgabewerte und deren Datentypen wird daraufhin die Antwort in die einzelnen Werte aufgeteilt. Schließlich endet der Befehl mit der Rückgabe der konvertierten Rückgabewerte des Gerätes.

Mit den zuvor genannten Daten kann die Grundfunktionalität der Datenschnittstelle erreicht werden, allerdings können die *Messskripte* weiterhin Fehler enthalten. Daher werden zusätzlich zu den genannten benötigten Daten, für die Grundfunktionalität, Daten für weitere Funktionalitäten der Datenschnittstelle hinterlegt.
Dies betrifft vor allem die Parameter, da diese ein *Messskript* unbrauchbar machen können, auch wenn die genutzten Geräte dieselben Funktionen unterstützt. Dies ist zum Beispiel der Fall, wenn ursprünglich eine Spannungsquelle auf 12 Volt eingestellt werden sollte und diese gegen ein anderes Modell ausgetauscht wird. Diese könnte zum Beispiel nur 10 Volt unterstützt. In diesem Fall würde das *Messskript* bezüglich des Aufbaus und Ablaufs weiterhin korrekt bleiben, allerdings kann es nicht die ge-

wünschten Resultate erbringen.
Für diesen Fall sollen den Parametern zusätzliche Informationen hinzugefügt werden können, wie zum Beispiel der Wertebereich bei numerischen Werten.

Wenn eine Funktion eines Gerätes abgerufen wird, sollen die Parameter gegen ihre möglichen Werte abgeglichen werden. Somit soll ein ungültiges *Messskript* erkannt werden, um die Messung an dieser Stelle als fehlerhaft zu markieren.
Die Einschränkungen des Parameters hängen dabei vom genutzten Datentypen ab. Zum Beispiel kann bei einem *String* die maximale Länge relevant sein, während bei einer Dezimalzahl das Minimum, Maximum und die möglichen Nachkommastellen relevant sein können.

4.2. Datenspeicher

In diesem Unterkapitel soll auf die Speicherung der Daten eingegangen werden. Dazu werden zunächst die nötigen Anforderungen an den Datenspeicher aufgestellt. Daraufhin wird in Abschnitt 4.2.2 eine Übersicht unterschiedlicher Datenbanktypen aufgestellt, sowie ihre Vor- und Nachteile für die zu speichernden Daten genannt. In Abschnitt 4.2.3 wird schließlich ein Datenbankmanagementsystem für die Speicherung der Daten ausgewählt.

4.2.1. Generelle Anforderungen

Der genutzte Datenspeicher muss mehrere Bedingungen erfüllen.
Er muss von unterschiedlichen Geräten aus erreichbar sein. Dies ist nötig, da die Labormitarbeiter unterschiedliche Messungen, von unterschiedlichen Computern aus, gleichzeitig ausführen können sollen. Dabei müssen die Daten für jeden Mitarbeiter konsistent bleiben.

Eine weitere Anforderung ist, dass die Daten permanent gespeichert werden. Sie sollen somit nicht nur auf eine einzelne Messung beschränkt sein.

Die gespeicherten Daten sollen zusätzlich möglichst widerstandsfähig gegen Fehler beim Speichern und Aktualisieren der Daten sein. Wenn zum Beispiel das Speichern eines neu hinzugefügten Gerätes aufgrund eines unerwarteten Fehlers fehlschlägt, soll dies die vorherigen Daten in einem weiterhin korrekten Zustand belassen.

Eine optionale Eigenschaft ist das Erstellen von Nutzergruppen. Dies ist zum Beispiel dafür relevant, dass ausschließlich der Laborverantwortliche, oder ausgewählte

Personen, die Befehle eines Gerätes hinzufügen oder anpassen können.

Diese genannten Anforderungen werden besonders durch Datenbanken erfüllt, weshalb mehrere unterschiedliche Datenbanktypen für die Speicherung der Daten betrachtet wurden. Auf diese wird im folgenden Abschnitt eingegangen.

4.2.2. Betrachtete Datenbanktypen

Generell ist es möglich, die Daten in jeder Art von Datenbanken zu speichern. Es existieren dennoch Datenmodelle, welche von unterschiedlichen Datenbanktypen besser umgesetzt werden können, als von anderen. Aus diesem Grund wurden für die gemeinsame Speicherung der Daten unterschiedliche Datenbanktypen betrachtet. Diese werden vor dem Hintergrund der zu speichernden Daten auf ihre Vor- und Nachteile untersucht und verglichen. Auf diese Datenbanktypen wird im Folgenden eingegangen.

SQL

SQL Datenbanken speichern die Daten in Tabellen und sind dabei besonders auf relationale Daten ausgelegt. Jede Tabelle wird mit einem festen Schema erstellt, welches für jede Spalte der Tabelle den Datentypen vorgibt. Für die Abfrage der Daten wird die *Structured Query Language* genutzt.

Für die in der Datenschnittstelle genutzten Daten bietet das feste Schema einer Tabelle jedoch einen Nachteil. Der Grund dafür liegt darin, dass das Schema abhängig von den gespeicherten Daten sein kann. Dies ist bei den Parametern und ihren Wertebereichen der Fall. Für einen numerischen Parameter werden zum Beispiel andere Einschränkungen, wie das Maximum und Minimum, gespeichert, als für einen *String*, welcher an das Gerät gesendet werden soll. Dieses Problem kann auf unterschiedliche Weisen gelöst werden.

Eine Möglichkeit ist, für jeden Datentypen eine eigene Tabelle zu erstellen, in welcher die möglichen Einschränkungen definiert werden können. Bei der Definition eines Parameters in einem der *SCPI*-Befehle werden dafür zwei Werte gespeichert. Der erste gibt an, um welche Art von Datentyp es sich handelt. Der Zweite ist ein Verweis auf die entsprechende Zeile der Tabelle, welche für den entsprechenden Datentypen vorgesehen ist.
Dieses Vorgehen hat mehrere Nachteile. Zum einen ist die Abfrage der Einschränkungen abhängig vom angegebenen Datentypen. Dies führt zu einer komplexen Abfrage mit vielen *Joins*. Außerdem müssen die Daten in der Anwendung zusammengefügt

werden, da sie nicht in einer einzelnen Abfrage sinnvoll ausgegeben werden können. Ein weiterer Nachteil ist, dass in der Datenbank auf der Spalte, welche die Referenz enthält, kein *Fremdschlüssel* definiert werden kann, da diese Spalte auf mehrere Tabellen verweisen kann. Es kann somit von der Datenbank nicht automatisch kontrolliert werden, dass der verwiesene Wert auch tatsächlich existiert.
Eine Realisierung der Datenschnittstelle durch dieses Vorgehen ist somit zwar möglich, würde allerdings durch den höheren Aufwand bei den Abfragen keine guten Ergebnisse erzielen.

Eine weitere Möglichkeit ist, alle möglichen Einschränkungen für jeden Datentypen in einer großen Tabelle zu speichern und die für einen Parameter irrelevante Werte auf `null` zu belassen. Dies ist jedoch ebenfalls keine gute Lösung, da ein Großteil der Tabelle leer sein wird und es außerdem nicht möglich ist, unterschiedliche Bedingungen an Wertelisten zu stellen. Es wäre zum Beispiel nicht möglich, in einer Liste von Werten anzugeben, dass der erste Wert einen anderen Wertebereich hat, als die darauf folgenden. Daher ist dies ebenfalls keine gute Lösung.

Eine dritte Möglichkeit wäre es, die Wertebereiche in einem „Kompositfeld“ wie *JSON* oder *XML* anzugeben, welche von vielen SQL Datenbanken unterstützt werden. Dabei handelt es sich häufig nur um einen langen *String*, welcher gespeichert werden kann. Dies kann jedoch als eine Verletzung der ersten Normalform angesehen werden, da die Daten der Wertebereiche nicht *atomar* in der Datenbank gespeichert werden. Außerdem müssten diese Daten in der Datenschnittstelle ebenfalls nach dem Auslesen noch verarbeitet werden. Daher ist auch bei dieser Umsetzung von keinen guten Resultaten auszugehen.

Graphen

Graphendatenbanken zählen zu den *NoSQL (Not only SQL)* Datenbanken. Unter *NoSQL* sind Datenbanken zusammengefasst, welche nicht auf einem relationalen Schema basieren. Sie sind somit unter anderem auch *schemafrei*.
Graphenbasierte Datenbanken nutzen *Knoten* und *Kanten* zur Speicherung der Daten. Die Knoten stellen die zu speichernden Daten dar, während die Kanten zwischen zwei Knoten die Beziehung untereinander darstellen. Sie eigenen sich daher vor allem, um stark vernetzte Daten darzustellen und wie diese miteinander in Verbindung stehen.
Anders als bei *SQL* Datenbanken existiert bisher noch kein genereller Standard als Abfragesprache. [9, 10]

Für eine Realisierung der Datenschnittstelle können graphenbasierte Datenbanken genutzt werden. Dafür werden die Geräte, Funktionen und auch Parameter durch Knoten realisiert, welche daraufhin durch Kanten miteinander verbunden werden.

Da die Datenbanken *schemafrei* sind, können auch spätere Erweiterungen um weitere Verbindungsmethoden, wie *JTAG* oder *SPI*, einfach hinzugefügt werden. Dafür müssen nur die neuen Befehlsarten als Knoten hinzugefügt werden. Die Zuweisung der neuen Befehle, zu den Geräten, kann über eine neue Kante realisiert werden. Die bisherige Struktur der Datenbank bleibt davon unbetroffen.
Graphendatenbanken bieten jedoch auch Nachteile. Dazu zählt vor allem das Fehlen einer einheitlichen Sprache für die Erstellung der Abfragen. Somit ist die Wahl des Datenbankmanagementsystems auch von den genutzten Programmiersprachen abhängig, da ein spezifischer Treiber für diese Programmiersprache existieren muss.
Das Fehlen eines Schemas kann ebenfalls ein Nachteil sein. Die gespeicherten Daten müssen keinem Schema entsprechen und können daher inkompatibel mit der Datenschnittstelle sein, wenn sie zuvor falsch in der Datenbank gespeichert wurden. Daher müssen die Daten beim Auslesen und Eintragen auf ihre Korrektheit überprüft werden, was zu einer zusätzlichen Komplexität bei der Verwaltung der Daten führt.

Dokumentenbasiert

Dokumentenbasierte Datenbanken zählen ebenfalls zu den *NoSQL* Datenbanken und sind daher auch *schemafrei*. In ihnen werden die Daten in einzelnen Dokumenten abgespeichert. Jedes dieser Dokumente erhält einen eindeutigen Identifikator, wodurch die Dokumente referenziert werden können. Ein Dokument kann zum Beispiel Text, binäre Daten oder auch *Key-Value-Paare* enthalten und somit die Daten, innerhalb des Dokumentes, strukturiert ablegen.
Auch für die dokumentenbasierten Datenbanken existiert keine generell genutzte Sprache für die Abfrage der Daten. [11, 12]

Durch die *schemafreie* Speicherung der Daten hat eine dokumentenbasierte Datenbank dieselben Vorteile und Nachteile wie auch die Graphendatenbank. Die Vorteile waren die einfache Erweiterung und das Speichern unterschiedlicher Daten, ohne diese in separaten Dokumenten bzw. Knoten speichern zu müssen. Die Nachteile umfassen einen hören Aufwand bei der Datenkontrolle, da diese keinem festen Schema folgen muss wie bei *SQL* Datenbanken.

4.2.3. Auswahl des Datenbankmanagementsystems

Aufgrund der in Abschnitt 4.2.2 genannten Vor- und Nachteile werden sowohl dokumentenbasierte, als auch graphenbasierte Datenbanken in Betracht gezogen. *SQL* Datenbanken werden aufgrund des festen Schemas für ihre Tabellen und den damit verbundenen Problemen ausgelassen, auch wenn sie die Daten ebenfalls speichern können.

Für Graphendatenbank wird *Neo4j* betrachtet. Dieses ist eine *Open-Source* Datenbankmanagementsystem für Graphendatenbanken, welche die selbst entwickelte Abfragesprache *Cypher* nutzt.[1]
Für Dokumentendatenbanken wird *MongoDB* betrachtet. Diese ist ein *Open-Source* Datenbankmanagementsystem für Dokumentendatenbanken, welches die Daten in *Key-Value-Paaren* und einem *JSON* ähnlichen Format speichert.[2]

Beide Datenbankmanagementsysteme sind wegen ihrer großen Nutzerbasis und der hohen Anzahl an Treibern für unterschiedliche Programmiersprachen gut für die Umsetzung der Datenschnittstelle geeignet.

Aufgrund der zuvor genannten Vorteile wurde sich schließlich für eine Umsetzung mit *MongoDB* als Datenbankmanagementsystem entschieden. Dies hat mehrere Gründe. Der Hauptgrund für die Wahl von *MongoDB* ist, dass die Dokumente *Schemafrei* sind und daher eine beliebige Form annehmen können, aber trotzdem *Schema-Validatoren* von *MongoDB* unterstützt werden. Diese können genutzt werden, um das Format der gespeicherten Dokumente beim Einfügen und beim Ändern gegen ein vorab festgelegtes Datenschema zu kontrollieren. Es kann somit sichergestellt werden, dass die Daten in einem Format vorliegen, welches von der Datenschnittstelle erwartet wird.
Der Unterschied der *Schema-Validation* zu einem festen Schema, wie bei *SQL* Datenbanken, liegt darin, dass mehrere Schemata für einen Wert in den Dokumenten zugelassen sein können, was bei einer *SQL* Datenbank nicht möglich ist.
Dies ist vor allem beim Speichern der Parameter und ihrer Wertbereiche relevant, da das Schema der Parameter abhängig vom gewählten Datentypen ist.
Auf die *Schema-Validation* wird im Kapitel Kapitel 5 auf Seite 31 noch weiter eingegangen.[13]

Ein weiterer Grund ist, dass die Daten nur wenige Relationen untereinander aufweisen, weshalb die Vorteile einer Graphendatenbank zur Abfrage der Relationen gering ausfallen.

Des Weiteren unterstützt *MongoDB ACID* Transaktionen. Die Abkürzung *ACID* ist in Tabelle 4.1 beschrieben.[14]

[1] Weitere Informationen über *Neo4j* können unter folgendem Link gefunden werden: `https://neo4j.com/`

[2] Mehr über *MongoDB* kann auf der folgenden Website erfahren werden: `https://www.mongodb.com/`

Name	Beschreibung
Atomicity	Transaktionen werden wie eine logische Operation betrachtet. Während der Transaktion müssen alle Änderungen erfolgreich sein, ansonsten werden alle vorherigen Änderungen rückgängig gemacht.
Consistency	Die Datenbank ist zu jeder Zeit in einem konsistenten und korrekten Zustand. Dies gilt auch für unerwartete Ausfälle, wie zum Beispiel einem Stromausfall oder defekter Hardware.
Isolation	Mehrere Transaktionen laufen voneinander unabhängig ab und beeinflussen sich nicht gegenseitig.
Durability	Die Daten werden vor Abschluss einer Transaktion auf einem persistenten Speichermedium gespeichert. Somit sind die Daten auch nach der Beendigung der Transaktion vorhanden.

Tabelle 4.1.: *ACID* Transaktionen [14]

Diese Eigenschaft ist besonders relevant, wenn Befehle nachträglich bearbeitet werden. Dabei kann es sonst zu Fehlern kommen, in denen nur Teile eines Befehls geändert wurden, weswegen dieser nicht mehr von den betroffenen Geräten korrekt ausgeführt werden kann. Außerdem haben unerwartete Fehler, wie ein Abbruch der Verbindung, beim Speichern der Daten somit keinen Einfluss auf die vorherigen gespeicherten Daten.

MongoDB unterstützt ebenfalls das Erstellen von Nutzergruppen. Diesen Gruppen oder *Rollen* können unterschiedliche Rechte auf den Datenbanken eingeräumt werden. Es ist somit möglich, ausschließlich zum Beispiel dem Laborverantwortlichen die Rechte zum Hinzufügen oder Ändern eines Gerätes zu geben, während alle Mitarbeiter ausschließlich Zugriff auf die Definitionen der Befehle haben, welche zur Ausführung der *Messskripte* nötig sind.
Dies ist auch relevant, wenn externe Partner außerhalb des ZEA-2 die Schnittstelle nutzen wollen. In diesem Fall sollen die Partner die Daten ebenfalls nicht ändern können.
Einzelnen Nutzern können ebenfalls spezielle Rechte, unabhängig ihrer Rolle, gegeben werden.[15]

Der letzte Grund ist die große Auswahl an Treibern für unterschiedliche Programmiersprachen. Eine Verbindung der Datenschnittstelle zur Datenbank ist für das Auslesen der Geräteinformationen und der *SCPI*-Befehle nötig. Durch die große Auswahl an Treibern kann die Datenschnittstelle in vielen verschiedenen Programmiersprachen umgesetzt werden. Dadurch vergrößert sich die Anzahl an möglichen Programmiersprachen für die Erstellung der *Messskripte*.[16]

Die Entscheidung *MongoDB* als Datenbankmanagementsystem zu nutzen ist auch in Absprache mit Klara Schnorrenberg getroffen. Für ihren Teil des *Laborsetups* sind

ebenfalls die *Schema-Validatoren* relevant, damit auch ihre gespeicherten Dokumente auf ein festgelegtes Schema kontrolliert werden können. Ihre gespeicherten Daten enthalten außerdem nur eine geringe Anzahl an Relationen untereinander, weshalb sie ebenfalls nicht von den Vorteilen einer graphenbasierten Datenbank, für die Erstellung von komplexen Netzen, profitiert.

5. Speicherkonzept in *MongoDB*

In diesem Kapitel wird auf das Konzept für die Datenbank eingegangen. Als Datenbankmanagementsystem wird *MongoDB* genutzt. Im ersten Unterkapitel werden die einzelnen Kollektionen, welche für die Speicherung der Daten vorgesehen sind, erläutert. In Unterkapitel 5.2 wird daraufhin auf die *Schema-Validatoren* eingegangen, welche die Kollektionen auf ihr Format kontrollieren. Unterkapitel 5.3 befasst sich mit der Validierung der gespeicherten Daten. Schließlich wird in Unterkapitel 5.4 das Auslesen der Daten beschrieben.

5.1. Kollektionen

In *MongoDB* werden die Daten in *Datenbanken* gespeichert. In jeder *Datenbank* werden dann beliebig viele *Kollektionen* erstellt. Eine Kollektion enthält einzelne Dokumente und ist somit vergleichbar mit einer Tabelle aus den *SQL* Datenbanken. Sie kann beliebig viele Dokumente enthalten, welche *schemafrei* sind. Innerhalb einer Kollektion werden die Dokumente im *BSON*-Format gespeichert. *BSON* ist eine Erweiterung und ein *Superset* von *JSON*. Es nutzt eine binäre Repräsentation zum Speichern der Daten und unterstützt außerdem eine größere Anzahl an Datentypen als *JSON*, wie zum Beispiel ein Datum oder Binärdaten.[17]

In *MongoDB* erhält jedes Dokument einen eindeutigen Schlüssel. Dieser wird innerhalb der Dokumente unter dem Wert „`_id`“ gespeichert. Über diesen Schlüssel ist es zum Beispiel möglich, innerhalb eines Dokuments auf ein anderes zu verweisen. Dafür kann einfach die „`_id`“ des Dokumentes angegeben werden, auf welches verwiesen wird. Eine weitere Möglichkeit dafür sind sogenannte *DBRef* Einträge. Bei diesen handelt es sich um ein eingebettetes Dokument, welches die Felder „`$id`“ und „`$ref`“ vorschreibt. In „`$id`“ wird der Fremdschlüssel gespeichert, während „`$ref`“ die Kollektion enthält, auf die verwiesen wird. Es kann ebenso noch das Feld „`$db`“ angegeben werden. Dies wird genutzt, um auf andere Datenbanken zu verweisen.

In einer Kollektion können ebenfalls *Indices* angelegt werden. Diese bestehen aus einem oder mehreren Feldern innerhalb der Dokumente. Ein *Index* ermöglicht eine

schnellere Abfrage der Daten, wenn der *Index* als Filterkriterium genutzt wird. Außerdem kann ein *Index* als *unique* markiert sein. In diesem Fall dürfen keine anderen Dokumente den gleichen Wert über den kompletten Index besitzen. Er ist somit ein weiterer Primärschlüssel für die Daten.

Zur Strukturierung der Datenbank sind insgesamt fünf Kollektionen vorgesehen:

1. `Devices`
2. `DeviceClasses`
3. `HighLevel`
4. `SCPI_Commands`
5. `SCPI_Command_Definitions`

Auf diese wird im Folgenden genauer eingegangen:

SCPI_Commands

Die Kollektion *SCPI_Commands* enthält alle *SCPI*-Befehle, welche von beliebigen Geräten unterstützt werden. Da die gleichen Befehle in vielen Geräten vorhanden sind, sich aber in ihrer Rückgabe und den Wertebereichen ihrer Parameter unterscheiden können, werden diese in einer eigenen Kollektion gespeichert. Dadurch können Redundanzen vermieden werden.

In den Dokumenten wird zunächst der eigentliche Befehl gespeichert. Dafür wird der Wert „`commands`“ genutzt. Er gibt das *Subsystem*, oder den *Common-Command* an und legt ebenfalls die Namen der Parameter fest. Die Parameter werden durch „Platzhalter“ im Befehl festgelegt. Jeder dieser „Platzhalter“ erhält einen eindeutigen Namen und wird nach dem Schema „<Parametername>“ eingetragen. Ein Beispiel für einen solchen Eintrag ist:

```
:SYSTem:TIME <Hours>,<Minutes>,<Seconds>
```

Die Position der Parameter wird durch die Reihenfolge der „Platzhalter“ festgelegt. Dieses Beispiel würde den *SCPI*-Befehl `:SYSTEM:TIME` mit den Parametern *Hours*, *Minutes* und *Seconds* definieren. Wenn ein Befehl an ein Gerät gesendet werden soll,

werden diese Platzhalter durch den Wert des Parameters ersetzt, welcher für den Namen oder die Position angegeben wurde.

Nicht jeder Parameter ist jedoch für das Ausführen der Befehle relevant. Daher werden ebenfalls die möglichen Kombinationen an Parametern gespeichert.
Zum Beispiel kann der eben gezeigte Befehl mit allen drei Parametern, nur mit *Hours* und *Minutes*, oder nur mit *Hours* aufgerufen werden. Ein Aufruf mit *Minutes* und *Seconds*, aber ohne *Hours* wäre jedoch nicht möglich.
Die möglichen Kombinationen sollen über die Namen der Parameter gespeichert. Bei dem vorherigen Beispiel wäre dies:

```
[  ["Hours", "Minutes", "Seconds"], ["Hours", "Minutes"], ["Hours"]  ]
```

Schließlich werden die Datentypen der Parameter festgelegt. Jedem Parameternamen wird dazu der mögliche Datentyp zugewiesen. Dafür wird ein eingebettetes Dokument erstellt, welches durch den Wert „`datatype`“ den möglichen Datentypen angibt. Abhängig von den dort gewählten Typen werden dann noch zusätzliche Felder vorausgesetzt. Hier werden jedoch nicht die Wertebereiche der Parameter festgelegt, da diese abhängig vom Gerät sind.

Zusammenfassend soll die Kollektion die in Abbildung 5.1 dargestellten Felder enthalten.

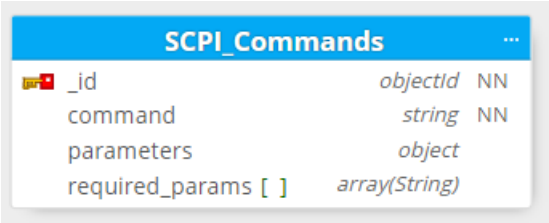

Abbildung 5.1.: *SCPI_Commands* Kollektion

Jedes Dokument enthält somit ein Feld „`command`“, welches den *SCPI*-Befehl angibt, sowie die Parameter mit ihren Datentypen und die benötigten Parameter. Das Feld „`command`“ muss immer angegeben werden.

Ein Eintrag für das zuvor genannte Beispiel kann durch das in Dokument 5.1 dargestellte Dokument realisiert werden.

```
1  {
2      "_id"         : null,
3      "command"     : ":SYSTem:TIME <Hours>,<Minutes>,<Seconds>",
4      "parameters" : {
5          "Hours"    : {"datatype": "uint"},
6          "Minutes" : {"datatype": "uint"},
7          "Seconds" : {"datatype": "double"}
8      },
9      "required_params" : [
10         ["Hours", "Minutes", "Seconds"],
11         ["Hours", "Minutes"],
12         ["Hours"]
13     ]
14 }
```

Dokument 5.1.: *SCPI* Definition zum Setzen der Systemzeit

In Zeile 3 wird festgelegt, welcher *SCPI*-Befehl definiert wird. In diesem Befehl sind bereits die „Platzhalter“ für die Parameter *Hours*, *Minutes* und *Seconds* angegeben. Die eigentliche Definition der Parameter geschieht in der Zeile 4. In diesem eingebetteten Dokument werden dann in den Zeilen 5 und 6 jeweils die Parameter *Hours* und *Minutes* definiert. Diese müssen dabei nicht negative ganzzahlige Werte sein. In Zeile 7 wird zusätzlich der Parameter *Seconds* als eine Dezimalzahl definiert. Dies ist zum Beispiel für das Angeben von Millisekunden relevant.
Schließlich werden in den Zeilen 10, 11 und 12 die unterschiedlichen Kombinationen der Parameter angegeben.
Das „`_id`“ Feld, in Zeile 2, wird beim Einfügen automatisch von *MongoDB* hinzugefügt und wurde daher hier nicht weiter spezifiziert.

HighLevel

Die *HighLevel* Kollektion enthält die Definitionen der Funktionen. Sie wird *HighLevel* genannt, da die Datenschnittstelle diese übergreifenden Funktionen bereitstellen soll, welche anschließend zu den entsprechenden *SCPI*-Befehlen umgewandelt werden.

Damit eine solche Funktion definiert werden kann, benötigt sie einen eindeutigen Namen, welcher in „`hl_cmd`“ gespeichert wird. Über diesen Namen wird sie später im *Messskript* aufgerufen. Zusätzlich zum Namen soll auch eine Beschreibung der Funktion hinterlegbar sein. Diese wird im Feld „`description`“ abgelegt.

Die *HighLevel*-Funktionen müssen ebenfalls Parameter und Rückgabewerte definieren.

Dafür wird in den Werten „`parameters`“ und „`response`“ der Name und der Datentyp der Parameter und Rückgabewerte definiert. Dabei handelt es sich um eingebettete Dokumente, in denen der Name der *Schlüssel* und die Definition der Datentypen der zugehörige *Wert* sind.

Da die Reihenfolge der Werte in den Dokumenten nicht immer gleich sein muss, wird die Reihenfolge der Parameter und Rückgabewerte separat gespeichert. Im Feld „`param_order`“ werden die Namen der Parameter in der Reihenfolge für den Aufruf angegeben. „`response_order`“ gibt die Reihenfolge der Rückgabewerte an.

Schließlich wird in „`required_params`“ gespeichert, welche Parameter für den Aufruf der Funktion notwendig sind. Diese sollten auch die ersten Werte in der Parameterreihenfolge sein. Der Wert wird, wie bei den *SCPI_Commands*, ebenfalls als doppeltes *String*-Array angegeben.

Abbildung 5.2 fasst die zuvor genannten Felder der Dokument in der *HighLevel* Kollektion noch einmal zusammen.

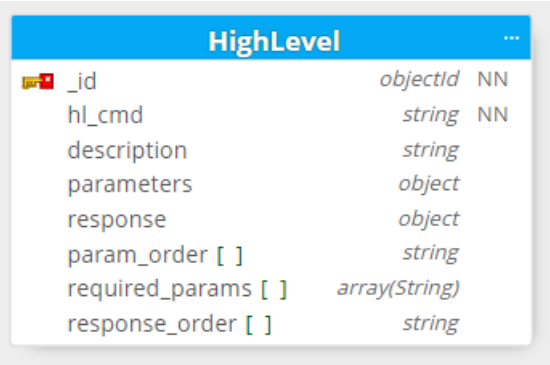

Abbildung 5.2.: *HighLevel* Kollektion

Die Dokumente enthalten somit einen Namen für die Funktion, sowie die Parameter für den Aufruf und die Rückgabewerte. Bei den Parametern und Rückgabewerten werden ebenfalls die Datentypen sowie ihre Reihenfolge gespeichert. Schließlich wird bei den Parametern angeben, welche von diesen vorausgesetzt werden.

Die *HighLevel*-Definition für das Setzen der Zeit, aus dem letzten Beispiel, kann wie in Dokument 5.2 dargestellt realisiert werden.

```
1  {
2      "_id"          : null,
3      "hl_cmd"       : "setSystemTime",
4      "description"  : "Setzt die Systemzeit auf die gegebenen Werte",
5      "parameters"   : {
6          "hour"   : {"datatype": "uint"},
7          "minute" : {"datatype": "uint"},
8          "second" : {"datatype": "double"}
9      },
10     "required_params" : [
11         ["hour", "minute", "second"],
12     ],
13     "param_order"     : ["hour", "minute", "second"],
14     "response"        : null,
15     "response_order"  : null
16 }
```

Dokument 5.2.: *HighLevel* Definition der „setSystemTime“ Funktion

Zeile 2 definiert den Funktionsnamen „setSystemTime“ unter welchem die Funktion nun genutzt werden kann. Die Zeilen 5 bis 9 definieren die Parameter der Funktion. In diesem Fall *hour* und *minute*, welche positive Ganzzahlen sein müssen, und *second*, welcher eine Dezimalzahl sein darf. Die Zeile 10f legt alle drei Parameter als benötigt fest und Zeile 13 legt die Reihenfolge dieser Parameter fest.
Die Felder „`response`“ (Zeile 15) und „`response_order`“ (Zeile 16) sind in diesem Fall ausgelassen, da von dem Befehl keine Antwort erwartet wird. Sie könnten auch komplett aus dem Dokument entfernt werden.

SCPI_Command_Definitions

In *SCPI_Command_Definitions* werden die konkreten Zuweisungen der Funktionen zu ihren *SCPI*-Befehlen gespeichert.

Dafür enthält das Dokument zunächst einen Verweis auf den *SCPI*-Befehl, welchen er umsetzt. In dem Wert „`command`“ wird dazu der „`_id`“ Wert des Dokumentes eingetragen, welches den Befehl in *SCPI_Commands* enthält. Über diesen „`_id`“ Wert kann dann der entsprechende *SCPI*-Befehl inklusive seiner Parameter ausgelesen werden.

Zusätzlich zum *SCPI*-Befehl enthält das Dokument auch einen Verweis auf eine *HighLevel*-Funktion. Diese Funktion ist dann mit dem gespeicherten *SCPI*-Befehl assoziiert. Hierbei wird der „`_id`“ Wert des Dokumentes aus der *HighLevel* Kollektion

in dem Feld „`hl`“ gespeichert.

Der „`command`“ muss immer angegeben werden, da die *SCPI*-Befehle immer ausgeführt werden können sollen, selbst wenn keine *HighLevel*-Funktion zugewiesen wurde. In diesem Fall soll die Funktion über die Angabe des *Program Headers* ausgewählt werden.

Da die eingetragenen Dokumente in der *SCPI_Command_Definitions* Kollektion die *HighLevel* und *SCPI_Commands* Einträge miteinander verbinden sollen, muss dies für jedes Gerät einzeln durchgeführt werden. Der Grund dafür ist, dass andere *SCPI*-Befehle für die Ausführung der Funktion nötig sein könnten.
Die Rückgaben der einzelnen *SCPI*-Befehle sind ebenfalls vom Gerät abhängig, daher werden sie erst in diesem Dokument angegeben. In dem Feld „`response`“ kann analog zur Definition des *SCPI*-Befehls in *SCPI_Commands* die Rückgabe des Befehls definiert werden.
Für die Abfrage der aktuellen Uhrzeit ist dies zum Beispiel:

```
<hour>,<minute>,<second>
```

Beim Auslesen der Antwort wird diese dann an den angegebenen Trennzeichen, hier den zwei Kommata, getrennt und an die Namen der Rückgabewerte gebunden.

Die Rückgabe der Befehle wird als *Bytes* eingelesen. Daher müssen die Datentypen der Rückgabewerte angegeben werden. Diese werden in dem eingebetteten Dokument „`response_types`“ angegeben. Der *Key* ist dabei der Name des Rückgabewertes, mit der Datentypdefinition als *Value*. Anhand der angegebenen Datentypen sollen dann die gelesenen *Bytes* konvertiert werden.

Die Einschränkungen an die Datentypen sind ebenfalls geräteabhängig. Für jeden Parameter kann in dem eingebetteten Dokument „`constraint`“ eine Einschränkung an die angegebenen Datentypen hinterlegt werden.

Schließlich ist es auch möglich, dass manche Befehle vor der angegebenen Funktion ausgeführt werden müssen. Das ist zum Beispiel der Fall, wenn das Format für die Ausgabe der Werte vorher gesetzt werden muss. Diese Befehle können in dem Array „`preceding_commands`“ angegeben werden. Sie werden dann an das Gerät vor dem eigentlichen Befehl gesendet. Die Befehle dürfen dabei keine weiteren Parameter oder Rückgaben enthalten.

Diese Einstellungen würden für die Grundfunktionalitäten der Datenschnittstelle ausreichen. Allerdings gibt es noch weitere Einstellungen, welche nötig sein könnten. Auf diese wird im Folgenden näher eingegangen.

Die Namen der Parameter können sich zwischen der Definition des *SCPI*-Befehls in *SCPI_Commands* und den Parametern in der *HighLevel*-Funktion unterscheiden. Es ist daher nötig, dass die Parameter miteinander verbunden werden. Daher werden in dem eingebetteten Dokument „`mapping`“ den *SCPI*-Parametern ihre zugehörigen Funktionsparameter zugewiesen. Wenn die Namen identisch sind, sollen diese aus der Zuweisung gelassen werden können.
Für die zuvor gezeigten Beispiele wäre diese Zuweisung von *hour* zu *Hours*, *minute* zu *Minutes* und *second* zu *Seconds*.

Es kann außerdem der Fall sein, dass die Daten in unterschiedlichen Einheiten vorliegen. Zum Beispiel erwartet ein Gerät die Daten in `Millivolt`, während die *HighLevel*-Funktion diese in `Volt` erwartet. In diesem Fall müssen die Werte vor dem Senden an das Gerät konvertiert werden. Dafür kann in dem eingebetteten Dokument „`conversions`“ für einen Parameter ein Konvertierungsfaktor angegeben werden, welcher vor dem Senden an das Gerät angewandt wird. Dies ist nur bei numerischen Werten möglich.
Dieses Vorgehen ist auch bei den Rückgabewerten möglich. Dabei werden die Faktoren in dem Dokument „`response_conversions`“ angegeben. Diese Konvertierungen werden vor der Rückgabe aus der *HighLevel*-Funktion angewandt und die Daten sind somit immer in denselben Einheiten vorhanden.

Ähnlich wie bei manchen Programmiersprachen soll auch mit Defaultwerten gearbeitet werden können. Diese können in dem eingebetteten Dokument „`param_defaults`“ angegeben werden.
Ebenso bietet der *SCPI*-Befehl nicht immer alle Rückgabewerte, welche von einer *HighLevel*-Funktion erwartet werden. In diesem Fall können diese analog in dem Dokument „`response_defaults`“ angegeben werden.

Die folgende Abbildung 5.3 stellt die *SCPI_Command_Definitions* Kollektion mit ihren Referenzen noch einmal dar.

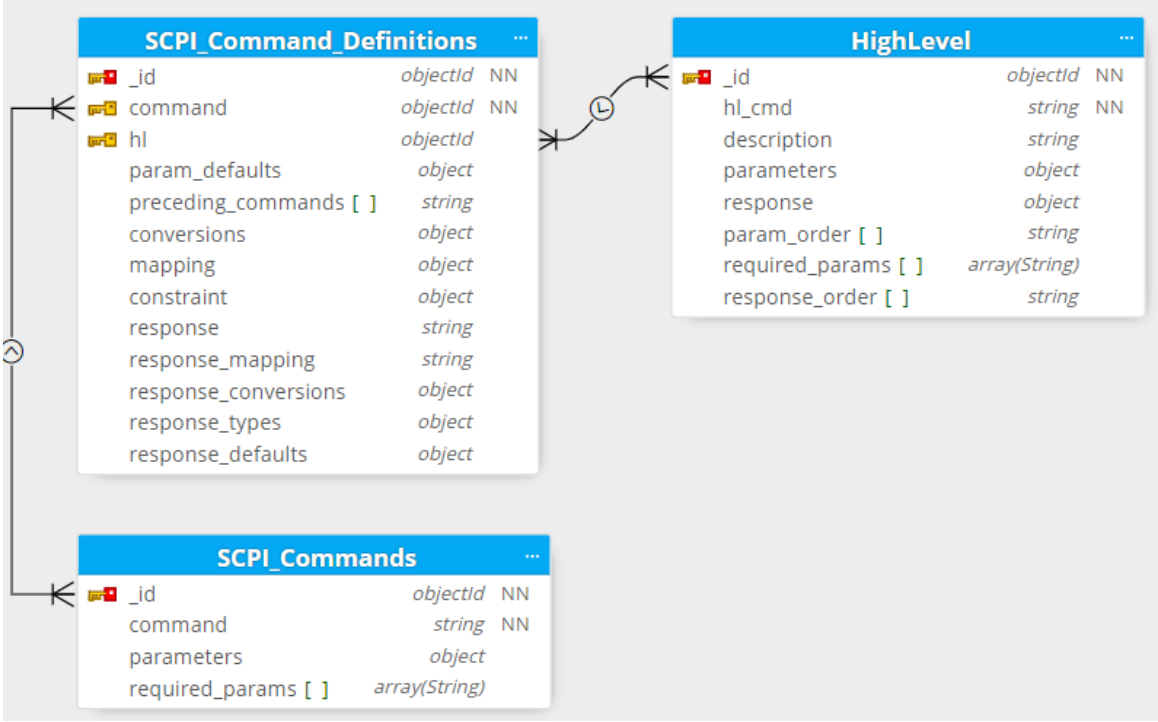

Abbildung 5.3.: Konzept der SCPI_Command_Definitions Kollektion und Referenzen

Die Dokumente der *SCPI_Command_Definitions* Kollektion verbinden somit die Dokumente der *SCPI_Commands* Kollektion mit den Funktionen der *HighLevel* Kollektion, durch die Referenzen in „`command`“ und „`hl`“. Zusätzliche Befehle welche vor der Ausführung des *SCPI*-Befehls ausgeführt werden müssen, können ebenfalls angegeben werden. Es werden zusätzlich das Format der Antwort auf den *SCPI*-Befehl und deren Datentypen bestimmten. Schließlich werden die Parameter und Antworten zueinander zugewiesen, durch mögliche *Defaultwerte* ergänzt und auf die gleiche Einheit umgerechnet.

Die beiden vorherigen Beispiele aus Dokument 5.1 und 5.2 können durch das in Dokument 5.3 dargestellte Dokument kombiniert werden.

```
1  {
2      "_id"     : null,
3      "command" : "<_id des SCPI Befehls>",
4      "hl"      : "<_id des HighLevel Befehls>",
5      "mapping" : {
6          "Hours"   : "hour",
7          "Minutes" : "minute",
8          "Seconds" : "second"
9      }
10 }
```

Dokument 5.3.: *SCPI_Command_Definitions* Dokument zur Verbindung des *SCPI* und *HighLevel* Befehls

Die Zeile 3 definiert den Wert „`command`“ welcher die Referenz auf den *SCPI*-Befehl enthält. Zeile 4 enthält wiederum in dem Feld „`hl`“ die Referenz auf den *HighLevel* Befehl. Diese beiden Werte werden bei der Speicherung des Dokumentes erstellt und sind daher in diesem Beispiel nicht eingetragen.
Beide Befehle nutzen jedoch unterschiedliche Namen für die Parameter. Daher werden diese in den Zeilen 5 bis 9 einander zugewiesen. In diesem Fall müssen die Parameter der *HighLevel*-Funktion (*hour*, *minute* und *second*) den Parametern des *SCPI*-Befehls (*Hours*, *Minutes* und *Seconds*) zugewiesen werden.

DeviceClasses

Die Dokumente der *DeviceClasses* Kollektion definieren *Geräteklassen*, welche einem Gerät zugewiesen werden können. Jede Klasse erhält einen Namen, welcher diese eindeutig identifiziert. Außerdem können Funktionen festgelegt werden, welche von jedem Gerät einer *Geräteklasse* umgesetzt werden müssen. Darüber kann sichergestellt werden, dass wenn zum Beispiel zwei Spannungsquellen ausgetauscht werden, beide die Funktion unterstützen, welche für die *Geräteklasse* „Spannungsquelle“ hinterlegt wurden.

Die benötigten Befehle werden durch eine Referenz auf die *HighLevel*-Befehle realisiert. Ein Gerät, welches die *Geräteklasse* erfüllen soll, muss dann eine Definition für diese Funktion enthalten. Dafür wird in dem Feld „`required_commands`“ die Kategorie „`SCPI`“ erstellt. In diese werden dann die Schlüssel bzw. „`_id`“ Werte der *HighLevel*-Befehle eingetragen. Die Unterteilung in die „`SCPI`“-Kategorie ermöglicht es, dass zukünftig noch weitere Übertragungsprotokolle hinzugefügt werden können, ohne dass das bisherige Design der Kollektionen angepasst werden muss.

Die Dokumente der Kollektion besitzen den in Abbildung 5.4 dargestellten Aufbau.

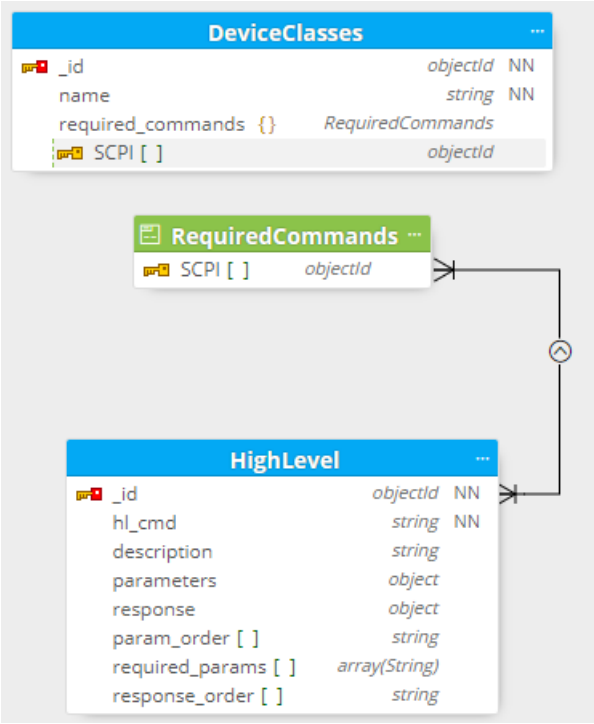

Abbildung 5.4.: DeviceClasses Kollektion und Referenzen

Die *DeviceClasses* Kollektion enthält somit Geräteklassen, welche einen eindeutigen Namen erhalten. Diese geben eine Liste von *HighLevel*-Funktionen vor, welche von den Geräten unterstützt werden müssen.
Das in 5.4 grün dargestellte Dokument *RequiredCommands* stellt das eingebettete Dokument „`required_commands`“ in *DeviceClasses* dar.
Dieses gibt die Art des Befehlsformates an. In diesem Fall ist dies *SCPI*, es sollen aber zukünftig noch weitere Übertragungsformate unterstützt werden, welche daraufhin in dieses Dokument eingetragen werden können.

Devices

Die Kollektion *Devices* soll die einzelnen Gerätemodelle enthalten. Jedes Gerät wird anhand seines Herstellers und seiner Modellbezeichnung eindeutig identifiziert. Diese bilden somit einen weiteren Primärschlüssel bzw. einen *unique Index*.

Durch eine Referenz in die *SCPI_Command_Definitions* Dokumente werden die vom Gerät unterstützten Befehle angegeben. Durch diese Referenz können die Befehle

an das Gerät angepasst werden, aber auch von Geräten einer zum Beispiel gleichen Modellreihe wiederverwendet werden.
Die Referenzen werden dabei in der Kategorie „`SCPI`“ gespeichert. Somit können zukünftig andere Übertragungsmethoden oder Befehlssätze hinzugefügt werden, ohne die bisherige Struktur der Daten ändern zu müssen. In diesem Array werden die „`_id`“ Werte der *SCPI_Command_Definitions* Dokumente gespeichert. Diese stellen die möglichen *SCPI*-Befehle des Gerätes dar.

Die in Abbildung 5.5 gezeigte Grafik stellt noch einmal das gesamte Konzept der Datenbank dar.

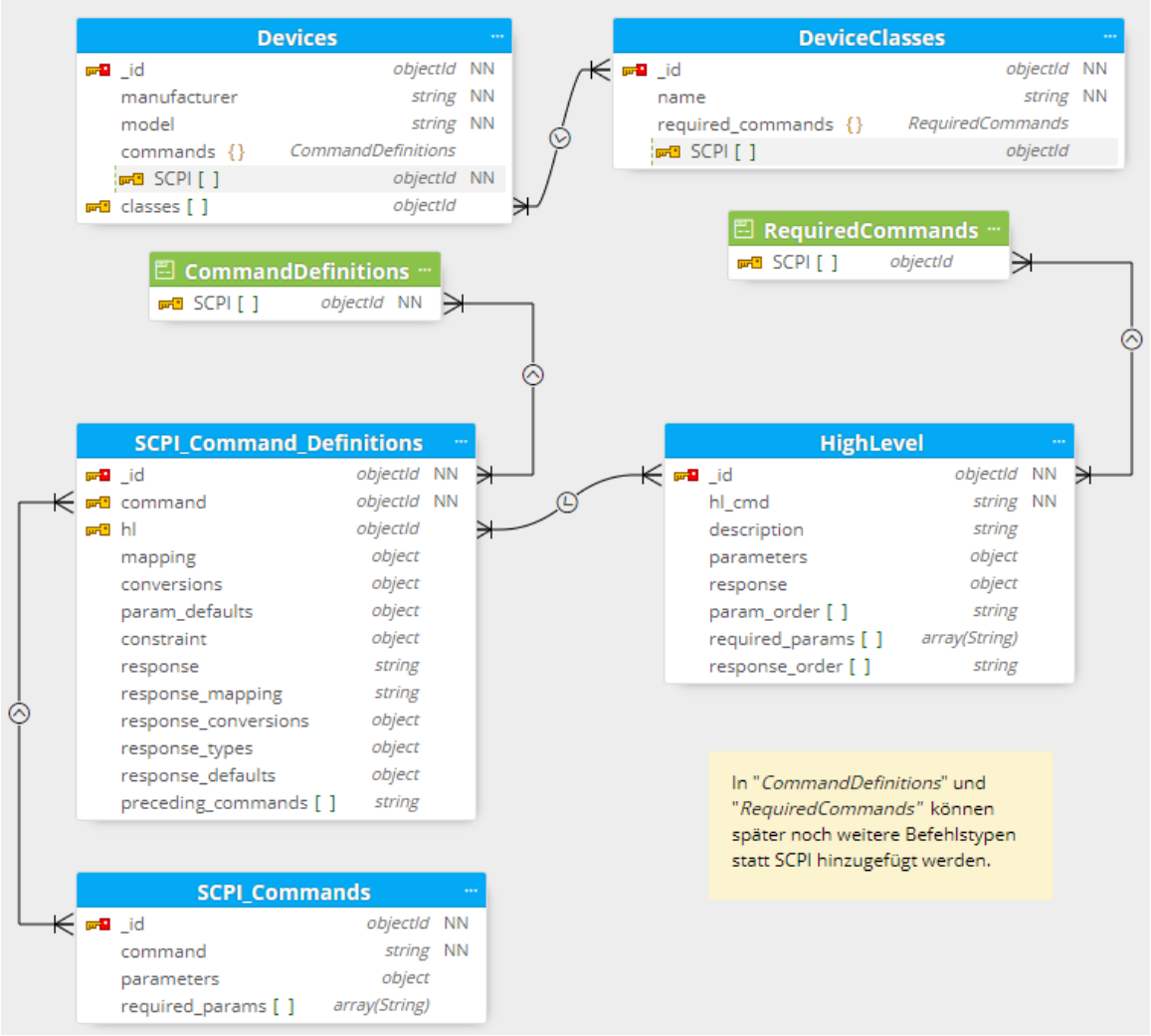

Abbildung 5.5.: Konzept der MongoDB Collections

In der Abbildung 5.5 sind die fünf Kollektionen, in Blau, mit ihren Referenzen dargestellt. Die in grün dargestellten Dokumente „CommandDefinitions“ und „Required-

Commands“ sind eingebettete Dokumente. „CommandDefinitions“ befindet sich in den Dokumenten der *Devices* Kollektion unter dem Wert „`commands`“, während „RequiredCommands“ in den Dokumenten der *DeviceClasses* Kollektion in dem Wert „`required_commands`“ gespeichert wird. Beide dieser eingebetteten Dokumente geben das genutzte Übertragungsformat an. Bisher ist dies nur *SCPI*, es soll allerdings auch noch die Möglichkeit geben, zukünftig weitere Formate hinzuzufügen. In diesem Fall würde diese in den eingebetteten Dokumenten eingetragen und somit müsste sich der generelle Aufbau der Kollektionen nicht ändern.

5.2. Schema-Validatoren

Schema-Validatoren sind eine Untergruppe der von *MongoDB* unterstützten Validatoren. Sie basieren auf dem vierten Entwurf des *JSON Schema* Standards. [13, 18, 19]

Das Thema der *Schema-Validation* ist für den Umfang dieser Arbeit zu komplex, um auf alle Kollektion und deren *Validatoren* einzugehen. Es wird daher anhand eines minimalen Beispiels die Funktionsweise der *Schema-Validation* demonstriert und das generelle Vorgehen in den Kollektionen daraufhin erläutert.

Die *Schema-Validation* basiert auf dem *JSON Schema*. Dieses definiert Operatoren, welche die Eigenschaften und die Struktur eines *JSON* Dokumentes festlegen. *MongoDB* hat diese Operatoren um weitere Optionen erweitert, um die zusätzlichen Datentypen der *BSON* Dokumente zu unterstützen. Die wichtigsten Operatoren sollen im Folgenden vorgestellt werden.

bsonType

Der Operator *bsonType* definiert, welcher Datentyp für ein Feld des Dokumentes erwartet wird. Es kann ein Datentyp oder eine Liste von möglichen Datentypen definiert werden. Wenn das Dokument in die Datenbank gespeichert wird, wird der Datentyp des Feldes mit den erlaubten Datentypen verglichen.

required

Der Operator *required* legt bestimmte Felder als Voraussetzung fest. Beim Speichern der Dokumente in die Datenbank wird kontrolliert, dass diese Felder in dem Dokument angegeben wurden.

properties

Der Operator *properties* kann ausschließlich für *Objekte*, also eingebettete Dokumente, genutzt werden. Mit ihm lassen sich die Felder des Dokumentes, sowie deren Schemata festlegen.

items

Der Operator *items* ist nur für Arrays verfügbar. Er erlaubt es, ein Schema für die Elemente des Arrays festzulegen.

Mit diesen vier Operationen lassen sich die Datentypen eines Dokumentes bereits festlegen.
Als Beispiel wird die Kollektion *Devices* aus Abbildung 5.5 durch einen *Schema-Validator* kontrolliert.
Dieser soll festlegen, dass es sich bei „`model`“ und „`manufacturer`“ um *Strings* handelt, während „`_id`“ vom Typen *ObjectId* ist.
„`classes`“ ist ein Array von Referenzen, daher soll dieses als Array von *ObjectId* Datentypen definiert werden.
Schließlich handelt es sich bei *commands* um ein eingebettetes Dokument. Dieses enthält den Wert *SCPI* welcher ebenfalls ein Array von Referenzen ist und somit ein Array von *ObjectIds* sein soll.
Die beiden Felder „`model`“ und „`manufacturer`“ sollen außerdem als vorausgesetzt angesehen werden.

Im Folgenden soll der Validator für dieses Dokument erläutert werden.
Da es sich um eine Dokumentendantenbank handelt, ist die oberste Ebene des Dokumentes ein *object* bzw. ein Dokument. Daher können diesem über den Operator *properties* die Felder vorgeschrieben werden. Zunächst werden die Felder „`_id`“, „`model`“ und „`manufacturer`“ erstellt. Dafür muss nur der Datentyp vorgeschrieben werden.
Ein Validator für diese Datentypen wird in Validator 5.4 dargestellt.

```
1  {
2    "properties": {
3      "_id": {
4          "bsonType": "objectId"
5      },
6      "manufacturer": {
7          "bsonType": "string"
8      },
9      "model": {
10       "bsonType": "string"
11     }
12   }
13 }
```

Validator 5.4.: Validator der Datentypen für „`_id`“, „`model`“ und „`manufacturer`“

Zeile 2 definiert den Operator *properties*. Über diesen lassen sich die Felder des Dokumentes bestimmen. In Zeile 3 wird daraufhin das Feld „`_id`“ definiert. Dieses soll durch das *Sub-Schema* `{"bsonType": "objectId"}` validiert werden (Zeilen 3 bis 5).

Der darin enthaltene Operator *bsonType* gibt dabei an, dass der einzige erlaubte Datentyp *ObjectId* ist.
Wenn nun ein Dokument gegen diesen Validator verglichen wird, muss das Feld „`_id`“, wenn es angegeben ist, vom Typen *ObjectId* sein.
Dasselbe Vorgehen wird in den Zeilen 6f und 9f genutzt, um den Feldern „`manufacturer`“ und „`model`“ die Datentypen *String* zuzuweisen.

Nun soll das Array „`classes`“ mit seinen Referenzen definiert werden. Der Wert „`classes`“ muss dafür den Datentypen Array haben, während die darin enthaltenen Objekte vom Typ *ObjectId* sein müssen. Ein solcher Validator ist in Validator 5.5 dargestellt.

```
1  {
2    "properties": {
3      "classes": {
4          "bsonType": "array",
5          "items": {
6              "bsonType": "objectId"
7          }
8      }
9    }
10 }
```

Validator 5.5.: Validator der Datentypen für „`classes`“

Zeile 3 definiert das Feld „`classes`“. Dieses ist in Zeile 4 als Datentyp „Array“ festgelegt und Zeile 5 definiert ein *Sub-Schema* für die Arrayelemente durch den Operator *items*. Diese Arrayelemente sind in Zeile 6 als *ObjectId* festgelegt. Somit muss das komplette Array, welches in „`classes`“ gespeichert wird, nur *ObjectIds* enthalten.

Schließlich soll noch das eingebettete Dokument „`commands`“ mit dem Feld „`SCPI`“ definiert werden. „`SCPI`“ enthält ebenfalls wie „`classes`“ die Referenzen und daher *ObjectIds*. Der Validator für dieses eingebettete Dokument ist in Validator 5.6 dargestellt.

```
1  {
2    "properties": {
3      "commands": {
4          "bsonType": "object",
5          "properties": {
6              "SCPI": {
7                  "bsonType": "array",
8                  "items": {
9                      "bsonType": "objectId"
10                 }
11             }
12         }
13     }
14   }
15 }
```

Validator 5.6.: Validator der Datentypen für „`commands`“

Die Zeilen 2f definieren das Feld „`commands`“ und legen fest, dass es sich dabei um ein eingebettetes Dokument handelt. Daher kann in Zeile 4 die Operation *properties* erneut genutzt werden, um die Felder des „`commands`“ Dokumentes festzulegen. Dafür wird in Zeile 6 das Feld „`SCPI`“ definiert, welches sich in dem Dokument „`commands`“ befinden soll. Die Definition der Datentypen für das Array von Referenzen ist daraufhin identisch wie bei Validator 5.5 in den Zeilen 7 bis 9 zu sehen ist.

Als Letztes soll festgelegt werden, dass die Felder „`model`“ und „`manufacturer`“ immer angegeben werden müssen. Dafür wird die Operation *require* genutzt. Diese erhält die Namen der benötigten Felder. Die Bedingung kann durch den in Validator 5.7 gezeigten Operator angegeben werden.

```
1  {
2      "require": ["model", "manufacturer"]
3  }
```

Validator 5.7.: Validator der vorgegebenen Felder „`model`“ und „`manufacturer`“

Die beiden Felder „`model`“ und „`manufacturer`“ werden somit in Zeile 2 als Voraussetzung angegeben.

Wenn diese *Schema-Validatoren* nun kombiniert werden, kontrollieren diese die Anforderungen an die Dokumente der *Devices* Kollektion. Der vollständige Validator ist in Validator 5.8 noch einmal dargestellt.

```
1  {
2    "$jsonSchema": {
3      "bsonType": "object",
4      "required": ["manufacturer","model"],
5      "properties": {
6        "_id": {"bsonType": "objectId"},
7        "manufacturer": {"bsonType": "string"},
8        "model": {"bsonType": "string"},
9        "classes": {
10         "bsonType": "array",
11         "items": {"bsonType": "objectId"}
12       },
13       "commands": {
14         "bsonType": "object",
15         "properties": {
16           "SCPI": {
17             "items": {"bsonType": "objectId"}
18           }
19         }
20       }
21     }
22   }
23 }
```

Validator 5.8.: *Schema-Validator* der *Devices* Kollektion

Der Operator „`$jsonSchema`“ (Zeile 2) gibt *MongoDB* bei der Erstellung des Validators an, dass es sich um ein *JSON Schema* handelt.

Eine grafische Darstellung des Validators ist in Abbildung 5.6 dargestellt.

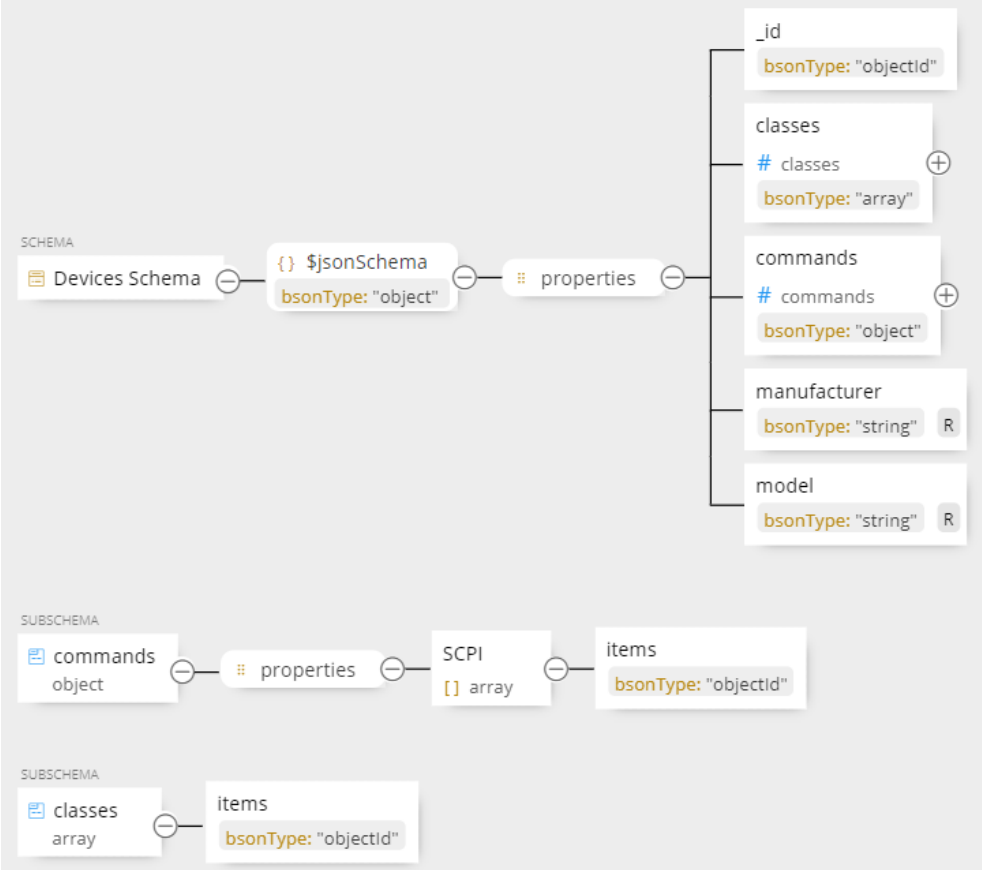

Abbildung 5.6.: Visualisierung des *Schema-Validators* für die *Devices* Kollektion

Die Abbildung zeigt das *Devices Schema*, welches die unterschiedlichen *properties* des Dokumentes definiert. Die einzelnen *Properties* und Operatoren werden dafür in einer baumartigen Struktur von links nach rechts dargestellt. Unter den Feldern werden jeweils die Optionen angezeigt, in diesem Fall ist das der *bsonType*, welcher den Datentypen angibt.
Die beiden *Properties* „`classes`“ und „`commands`“ in der oberen Hälfte verweisen auf die entsprechenden *Sub-Schemata* in der unteren Hälfte der Graphik. Das „`R`“ in den Feldern „`model`“ und „`manufacturer`“ markiert diese Felder als „required“.

Generelles Vorgehen

Für die Datenschnitstelle werden *Schema-Validatoren* für alle Kollektionen definiert. Diese Betrachten unter anderem die Datentypen, als auch, ob die nötigen Felder in den Dokumenten angegeben wurden. Dafür kann zum Beispiel auch angegeben werden, dass ein Feld als *required* markiert wird, sobald ein anderes Feld vorhanden ist. Dies ist unter anderem für die Angabe der Parameter relevant. Wenn zum Beispiel

der Wert „`parameter`“ angegeben wurde, müssen ebenfalls die Reihenfolgen der Parameter festgelegt werden. Daher muss in diesem Fall auch das Feld „`param_order`“ als *required* markiert werden.

Eine weitere Funktion, welche in dieser Arbeit nicht betrachtet wird, ist die Möglichkeit mehrere *Sub-Schema* anzugeben. In diesem Fall muss ein Feld genau einem, einem beliebigen, oder allen angegebenen Schemata entsprechen.
Durch diese Funktion sollen die unterschiedlichen Datentypen realisiert werden. Jede Definition eines Datentypen soll das Feld „`datatype`“ enthalten, welches den Datentypen des Parameters enthält. Über diesen Datentypen kann dann das korrekte *Sub-Schema* aus der Liste der *Schemata* identifiziert werden und die restlichen Felder der Parameterdefinition können ebenfalls kontrolliert werden. Falls der Parametertyp in „`datatyp`“ ungültig ist, wird keines der *Sub-Schemata* dem Dokument entsprechen und die gesamte Definition der Funktion oder des *SCPI*-Befehls ist ungültig.
Somit sollen die Formate der Kollektionen und deren Dokumente sichergestellt werden.[1]

5.3. Daten-Validation

MongoDB unterstützt nur die Validierung des Schemas, jedoch keine Validierung der Daten. Es können somit nicht alle Fehler in den Eingabedaten abgefangen werden. Zum Beispiel kann ein Eintrag eines *SCPI*-Befehls in den Feldern „`command`“ andere Parameter definieren, als in „`parameters`“ angegeben wurde.
Zum Beispiel würde das folgende Dokument 5.9 in der *SCPI_Commands* Kollektion die *Schema-Validierung* erfüllen, allerdings nicht die korrekten Daten enthalten.

```
{
    "command": "DISPlay:TEXT:DATA <Text>",
    "parameters": {
        "andererParameter" : {"datatype": "string"}
    },
    "required_params": [
        ["weitererParameter"]
    ]
}
```

Dokument 5.9.: Fehlerhafter Eintrag in der *SCPI_Commands* Kollektion

[1] Weitere Informationen über *Schema-Validation* können unter dem Link `https://www.mongodb.com/docs/v6.0/reference/operator/query/jsonSchema/` gefunden werden

In diesem Fall sind die Parameter mit unterschiedlichen Namen angegeben. In Zeile 2 ist dieser noch mit dem Namen „`Text`“ angegeben, während er in Zeile 4 als „`andererParameter`“ benannt ist und schließlich in Zeile 7 „`weitererParameter`“ genannt wird. Da alle Parameter unterschiedliche Namen nutzen, werden diese individuell betrachtet und können nicht als ein gemeinsamer Parameter angesehen werden. Dieser Fehler kann jedoch nicht von der Schema-Validation erkannt werden. Diese kann nur die Daten mit „statischen“ Daten vergleichen, welche bei der Definition der Schema-Validatoren bereits bekannt sind. Die „dynamischen“ Einträge, also die einzelnen gespeicherten Werte in dem Dokument, können jedoch nicht miteinander verglichen werden.

Daher müssen die eingetragenen Daten in einer anderen, externen Software validiert werden. Dafür soll die bereits vorgesehene Verwaltungssoftware für die Datenschnittstelle genutzt werden.
Eine weitere Möglichkeit ist das Erstellen eines externen Programms, welches die gesamte Datenbank auf fehlerhafte Einträge kontrolliert. Dieses kann nach jeder Änderung der Daten ausgeführt werden, um die Definitionen auf ihre Korrektheit zu überprüfen. Es könnte ebenfalls in einem regelmäßigen Abstand ausgeführt werden, wie zum Beispiel jede Nacht.
Falls bei der Kontrolle der eingetragenen Dokumente ein Fehler auffällt, sollen diese fehlerhaften Geräteinformationen an die Verantwortlichen der Datenschnittstelle gesendet werden. Somit soll sichergestellt werden, dass die fehlerhaften Eintrage der Daten nicht erst während der Nutzung im *Messskript* auffallen und so eine aktuelle Messung unbrauchbar machen können.

Beispiele für die zu kontrollierenden Daten sind, dass die Parameter identisch genannt wurden, oder auch dass die definierten Funktionen alle Parameter für den *SCPI*-Befehl abdecken. Dafür können entweder die Funktionsparameter oder *Defaultwerte* genutzt werden.

5.4. Auslesung der Daten

Damit die Geräte in den *Messskripten* von der Datenschnittstelle genutzt werden können, müssen die Informationen für die einzelnen Geräte aus der Datenbank abgefragt werden. Dafür soll das Modell und der Hersteller eines Gerätes in den *Messskripten* angegeben werden. Anhand dieser Angaben sollen die Gerätedefinitionen aus der Datenbank abgefragt werden. Diese Definitionen enthalten die unterstützten Funktionen für das gegebene Gerät, die *SCPI*-Befehle welche diese Funktionen umsetzen und auch welche Parameter benötigt werden.

In *MongoDB* werden zwei unterschiedliche Methoden zum Auslesen der Daten unter-

stützt: Die Operation *find* und *pipelines*.
Bei der *find* Operation werden einzelne oder mehrere Dokumente aus einer konkreten Kollektion abgefragt. Dabei kann zusätzlich ein Filter für die Abfrage angegeben werden. Die Dokumente der ausgewählten Kollektion werden dann mit diesem Filter verglichen und alle Dokumente, welche dem Filter entsprechen, werden als Ergebnis ausgegeben. Der Filter kann dabei einzelne Werte umfassen, Filteroperatoren enthalten, oder auch eine *Schema-Validation* sein.
Von den ausgegebenen Dokumenten können dann noch die Felder ausgewählt werden, welche in der Ausgabe enthalten sein sollen. Dies wird *Projektion* genannt. Die *find* Operation ist jedoch nicht in der Lage, die Dokumente anzupassen. Das heißt zum Beispiel, dass verwiesene Dokumente nicht mit übernommen werden.

Eine *Pipeline* hingegen ist eine Abfolge von Operationen, welche auf allen Dokumenten einer Kollektion ausgeführt wird. Es werden dafür zunächst temporäre Kopien der Dokumente in der Datenbank erstellt, auf welchen die einzelnen Schritte der *Pipeline* ausgeführt werden. Diese Schritte sind in der Lage, die temporären Dokumente zu bearbeiten und auch auf andere Kollektionen zuzugreifen. Am Ende der *Pipeline* können somit die erstellten Dokumente ausgegeben oder in eine andere Kollektion gespeichert werden.[2]

Damit nicht zu viele Abfragen für die einzelnen Befehle an die Datenbank gesendet werden, sollen die relevanten Daten in einem einzelnen Dokument zusammengefasst werden. Diese Daten sind jedoch über mehrere Kollektionen und Dokumente verteilt und auf diese Inhalte wird aus den Dokumenten referenziert. Daher soll eine *Pipeline* genutzt werden, welche diese Referenzen auflöst und alle Daten in einem Dokument zusammenfasst. Die *find* Operation reicht hierbei nicht aus, da diese keine Dokumente zusammenfügen kann.

Es werden insgesamt zwei *Pipelines* benötigt, um die Daten aus der Datenbank auszulesen. Die erste ist dabei für das Auslesen den Gerätedefinitionen zuständig. Dies umfasst die für das Gerät definierten Befehle, welche *SCPI*-Befehle für eine Funktion ausgeführt werden sollen, und auch die nötigen Parameter. Diese zweite *Pipeline* ist für das Auslesen der Geräteklassen zuständig. Dabei werden alle Funktionen ausgelesen, welche von einem Gerät unterstützt werden müssen, damit dieses die Geräteklassen erfüllt.
Auf diese *Pipelines* wird im Folgenden eingegangen.

[2]Eine Übersicht der *Pipeline* Operationen kann unter der folgenden URL gefunden werden: `https://www.mongodb.com/docs/manual/reference/operator/aggregation-pipeline/`

Gerätedefinitionen

Für die Datenschnittstelle bzw. das *Messskript* ist vor allem die *Devices* Kollektion relevant, da diese die Definition des gewünschten Gerätes enthält. Der Aufbau dieser Kollektion wurde in Unterkapitel 5.1 auf Seite 31 bereits erläutert. Anhand der ausgelesenen Daten können die definierten Funktionen für das Gerät, die zugehörigen *SCPI*-Befehle und auch die Parameter bestimmt werden. Die Geräteklassen sind für das Auslesen der Funktionen nicht relevant, weshalb diese in der *Pipeline* nicht betrachtet werden.
Im Folgenden soll nun auf die nötigen Schritte der *Pipeline* eingegangen werden, welche für das Auslesen der Daten und das Zusammenfügen der Dokumente nötig sind.

Die *Devices* Kollektion enthält zunächst die unterschiedlichen Dokumente zur Definition der Geräte. Das Schema der *Devices* Kollektion ist in Abbildung 5.7 gezeigt.

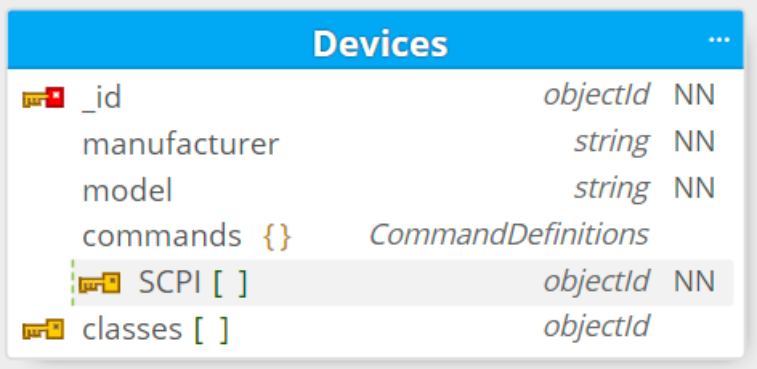

Abbildung 5.7.: Dokumentstruktur zu Beginn der *Pipeline*

Im ersten Schritt soll in der *Pipeline* nur das entsprechende Dokument für das aktuell betrachtete Gerät aus der Kollektion gefiltert werden. Grund hierfür ist, dass dadurch alle anderen Gerätedefinitionen nicht mehr die weiteren Schritte der *Pipeline* durchlaufen müssen und somit die gesamte Laufzeit der *Pipeline* reduziert wird. Für eine solche Abfrage ist das *$match* Statement geeignet. Dieses entfernt alle Dokumente aus der *Pipeline*, welche den übergeben Ausdruck nicht erfüllen.
Der *$match* Operator hat den in Operator 5.10 gezeigten Aufbau.

```
1 { "$match": { "<Feldname>": <Vergleichswert oder Ausdruck>, ... } }
2 # Oder
3 { "$match": <Boolescher Ausdruck auf dem Dokument> } }
```

Operator 5.10.: Format des *$match* Operators

Es gibt zwei Varianten für die Nutzung des Operators *$match*. Es können die einzelnen Feldnamen angegeben werden und für jeden dieser Werte kann entweder ein fester Wert, oder ein Vergleichsausdruck angegeben werden. Dieser Ausdruck kann zum Beispiel eine Wertespanne sein, oder ein Vergleich auf *größer* oder *kleiner*. Diese Variante ist in Zeile 1 gezeigt.
Alternativ kann ein Ausdruck angegeben werden, welcher zu einem booleschen Wert ausgewertet werden kann. Diese Variante ist in Zeile 3 gezeigt.[3]
Im Falle dieser *Pipeline* werden die Dokumente nach ihren Feldern „`manufacturer`“ und „`model`“ verglichen. Diese beiden Felder zusammen bilden einen Primärschlüssel, weshalb nur ein Dokument nach diesem Schritt noch in der *Pipeline* verbleibt.

Danach sollen die einzelnen Verweise innerhalb des Dokumentes aufgelöst werden. Der erste Verweis ist dabei in dem „`commands.SCPI`“ Array enthalten und verweist auf die gerätespezifischen Funktionen in der Kollektion *SCPI_Command_Definitions*.
Dafür wird die Operation *$lookup* genutzt. Diese Operation hat die in Operator 5.11 dargestellten Optionen.

```
1 {
2   "$lookup":
3     {
4         "from": <Kollektionsname>,
5         "localField": <Feld des Eingabedokumentes>,
6         "foreignField": <Verwiesenes Feld der 'from' Kollektion>,
7         "as": <Ausgabefeld>
8     }
9 }
```

Operator 5.11.: Format des *$lookup* Operators

In dem Feld „`from`“ (Zeile 4) wird die Kollektion angegeben, aus welcher die verwiesenen Dokumente gesucht werden sollen. Die Option „`localField`“ (Zeile 5) gibt an, in welchem Feld die Referenzen eingetragen sind, während das „`foreignField`“ (Zeile 6) angibt, auf welches Feld in der „`from`“ Kollektion verwiesen wird. Schließlich kann über die Option „`as`“ (Zeile 7) festgelegt werden, in welchem Feld die Ergebnisse gespeichert werden. Das Ergebnis ist dabei immer ein Array. Wenn es sich bei dem angegebenen Feld in „`localField`“ um ein Array handelt, werden alle Referenzen aus diesem Array aufgelöst und in dem „`as`“ Array ausgegeben. [4]

[3] Definition des *$match* Operators: `https://www.mongodb.com/docs/manual/reference/operator/aggregation/match/`
[4] Definition der *$lookup* Operation: `https://www.mongodb.com/docs/manual/reference/operator/aggregation/lookup/`

In diesem Fall wird das Feld „command.SCPI“ mit dem Feld *_id* in der Kollektion *SCPI_Command_Definitions* verglichen. Da es sich bei dem Feld um ein Array handelt, werden alle Referenzen ausgewertet und die Ergebnisse werden zurück in das Feld „`commands.SCPI`“ gespeichert.

Nach der Durchführung der vorherigen Schritte hat das Dokument die in Abbildung 5.8 gezeigte Struktur.

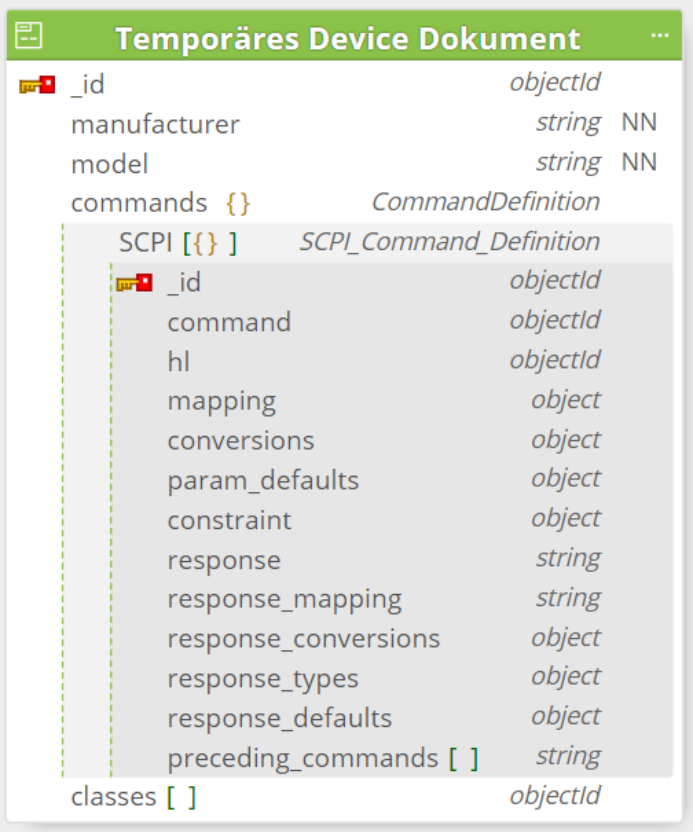

Abbildung 5.8.: *Devices-Pipeline* mit aufgelösten *SCPI_Command_Definitions* Verweisen

Die verwiesenen Dokumente aus der *SCPI_Command_Definitions* Kollektion sind nun in dem Feld „`command.SCPI`“ vorhanden. Jedes dieser Dokumente enthält die Referenzen auf einen *SCPI*-Befehl in dem Feld „`command`“ und auf die *HighLevel*-Funktion in „`hl`“. Diese Verweise sollen als nächstes aufgelöst werden.
Da es sich um ein Array handelt, muss dieses zuerst in einzelne Werte aufgeteilt werden. Dafür wird die Operation *$unwind* genutzt. Diese erstellt aus einem Array mit $\mathcal{N}$ Elementen $\mathcal{N}$ Dokumente, welche jeweils einen einzelnen Wert aus dem Array, in dem ursprünglichen Wert, enthalten. Dabei müssen leere Arrays ebenfalls betrachtet werden. In diesem Fall wird ein einzelnes Dokument mit einem *null* Wert erstellt.
Der *$unwind* Operator hat die in Operator 5.12 gezeigten Optionen.

```
1 {
2   "$unwind":
3     {
4       "path": "$<Feldname>",
5       "includeArrayIndex": "<Optionaler Feldname>",
6       "preserveNullAndEmptyArrays": <boolescher Wert>
7     }
8 }
```

Operator 5.12.: Format des *$unwind* Operators

Der *$unwind* Operator bekommt in dem Feld „`path`“ den Feldnamen des Arrays angegeben (Zeile 4). Das Feld „`includeArrayIndex`“ (Zeile 5) kann optional angegeben werden. In diesem Fall wird der Index des Elementes unter dem angegebenen Feldnamen gespeichert. Ansonsten wird der Index nicht mit gespeichert.
Schließlich gibt „`preserveNullAndEmptyArrays`“ (Zeile 6) an, ob bei einem leeren oder fehlenden Array ein *null* Wert gespeichert werden soll, oder ob das Dokument aus der *Pipeline* entfernt wird. Wenn dieser Wert nicht als `true` angegeben wird, wird das Dokument entfernt.[5]
In dieser *Pipeline* wird das Array „`commands.SCPI`“ in die einzelnen Werte zerlegt. Jedes Dokument enthält daraufhin ein verwiesenes Dokument aus der Kollektion *SCPI_Command_Definitions*.

Nun kann dasselbe Verfahren für das Auflösen der Referenzen über den Operator *$lookup* erneut angewandt werden.
Dafür werden die beiden Felder „`commands.SCPI.command`“ und „`commands.SCPI.hl`“ genutzt. Die erste gespeicherte Referenz „`commands.SCPI.command`“ wird in die Kollektion *SCPI_Commands* aufgelöst, während „`commands.SCPI.hl`“ in die *HighLevel* Kollektion aufgelöst wird.
Das Vorgehen ist hierbei identisch zu dem vorherigen *$lookup*, welches die Kollektion *SCPI_Command_Definitions* nutzt.

Die Ergebnisse dieser *$lookup* Operationen sind Arrays, welche ein einziges Dokument enthalten. Dies liegt daran, dass der Primärschlüssel „`_id`“ für die Verweise angegeben wird. Daher kann es nur ein Dokument geben, welches bei dem Verweis gefunden wird. Die beiden erhaltenen Arrays werden ebenfalls durch ein *$unwind* in ein einzelnes Dokument umgewandelt.

Nach dem Auflösen aller Verweise haben die Dokumente die in Abbildung 5.9 auf der nächsten Seite gezeigte Struktur.

[5] Definition des *$unwind* Operators: `https://www.mongodb.com/docs/manual/reference/operator/aggregation/unwind/`

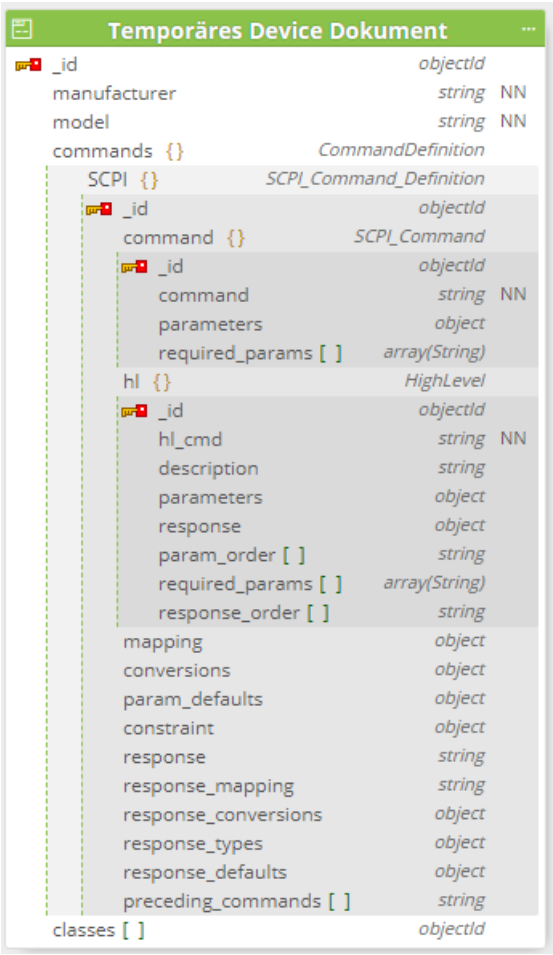

Abbildung 5.9.: *Devices-Pipeline* nach Auflösung aller Verweise

Das Feld „`commands.SCPI`“ enthält somit nun ein verwiesenes Dokument aus der *SCPI_Command_Definitions* Kollektion, welches wiederum die verwiesenen Dokumente aus den Feldern „`commands`“ und „`hl`“ enthält.

Es handelt sich allerdings noch um $\mathcal{N}$ Dokumente, wobei $\mathcal{N}$ die Anzahl der Befehle des Gerätes ist. Diese Dokumente enthalten jeweils einen Befehl. Daher werden diese im letzten Schritt in einem Dokument zusammengefasst, welches wieder die ursprüngliche Formatierung nutzt. Dabei enthielt „`commands.SCPI`“ ein Array von Verweisen. Nun soll dieser Wert das Array von eingebetteten Dokumenten enthalten, welche in den vorherigen Schritten der *Pipeline* aufgelöst wurden.
Für diese Operation kann *$group* genutzt werden. *$group* hat die in Operator 5.13 gezeigte Struktur.

```
1 {
2   "$group":
3     {
4       "_id": <Gruppierungskriterium>,
5       "<Feldname>": { <Gruppierungsvorschrift> : <Ausdruck> },
6       ...
7     }
8  }
```

Operator 5.13.: Format des *$group* Operators

Bei der Gruppierung werden die Dokumente nach bestimmten Kriterien wieder kombiniert. In dem Feld „`_id`“ (Zeile 4) werden die Gruppierungskriterien angegeben. Dafür werden ein oder mehrere Felder aus dem Dokument angegeben. Alle Dokumente werden anhand ihrer Werte in diesen Feldern daraufhin in Gruppen aufgeteilt. Anhand dieser Gruppen können dann die weiteren Felder des Ausgabedokumentes bestimmt werden. Dazu wird der Feldname angegeben und eine Gruppierungsvorschrift festgelegt. Schließlich wird dieser Vorschrift ein Wert zugewiesen, aus welchem sie die Informationen entnimmt. Dies kann ein Feld aus den aktuellen Dokumenten sein. Dieses Vorgehen ist in Zeile 5 gezeigt.
Die Gruppierungsvorschrift gibt an, wie die Daten aus den Feldern verarbeitet werden. Zum Beispiel kann mit der Vorschrift *$first* der Wert aus dem ersten Dokumente einer Gruppe übernommen werden, während *$push* alle Werte in einem Array zusammenfasst. Das Ergebnis wird in dem angegebenen Wert „`Feldname`“ gespeichert.[6]

Für diese *Pipeline* kann „`_id`“ als Gruppierungskriterium genutzt werden, da jedes Dokument noch den ursprünglichen „`_id`“ Wert des Dokumentes enthält. Die Felder „`model`“, „`manufacturer`“ und „`classes`“ sind ebenfalls in allen $\mathcal{N}$ Dokumenten identisch, sodass hier nur der Wert aus jeweils einem der Dokumente übernommen werden muss.
Die einzelnen aufgelösten Verweise sind aktuell in dem Wert „`commands.SCPI`“ gespeichert. Diese Werte werden nun durch *$group* alle in ein Array zusammengefasst und in einem temporären Wert gespeichert. Der Grund dafür ist, dass *$group* die Ergebnisse nicht in dem eingebetteten Dokument „`command`“ speichern kann.
Um dies zu umgehen, wird der temporäre Wert in dem nächsten Schritt der *Pipeline* durch ein *$addFields* wieder in das Dokument „`commands`“ eingefügt. Der letzte Schritt der *Pipeline* ist es, den temporären Wert zu entfernen. Dafür wird, wie beim *find*, die Operation *$project* genutzt. In dieser wird der temporäre Wert wieder entfernt.

[6] Die Definition der *$group* Operation sowie der Gruppierungsvorschriften: `https://www.mongodb.com/docs/manual/reference/operator/aggregation/group/`

Auf den Aufbau der *$addFields* und *$project* Operationen wird nicht näher eingegangen.

Nach dem Durchlauf der *Pipeline* sind alle Verweise des *command* Dokumentes aufgelöst und die Definitionen der Befehle liegen im Wert „`commands.SCPI`“ für die Datenschnittstelle vor.

Das finale Format der Dokumente ist in Abbildung 5.10 dargestellt.

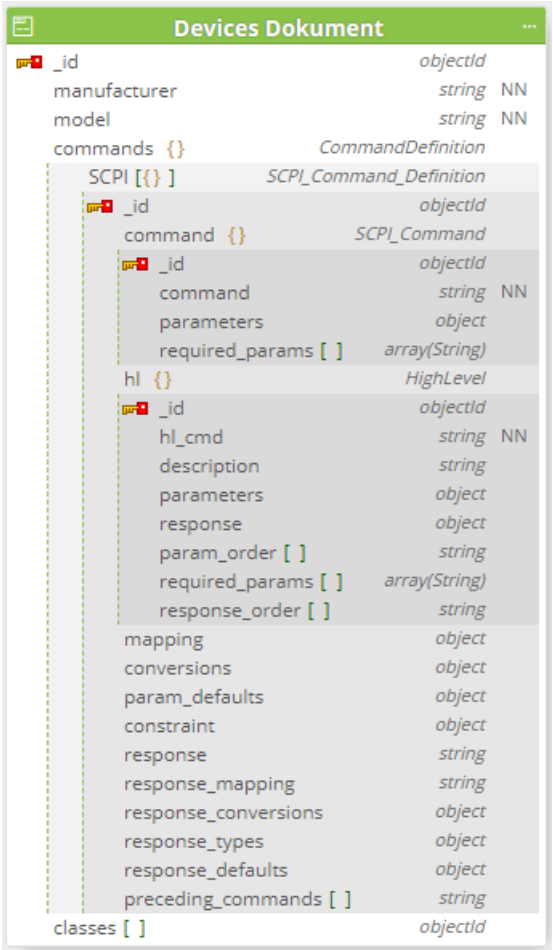

Abbildung 5.10.: *Devices-Pipeline* Ergebnis

Die vollständige Definition der *Pipeline* kann im Anhang A.1 gefunden werden.

Geräteklassen

Damit ein Gerät eine Geräteklasse erfüllt, muss es einen vordefinierten Satz an Funktionen realisieren. Diese Funktionen werden in den zugewiesenen Geräteklassen angegeben und sollen durch die folgende *Pipeline* ausgelesen werden.

Eine Gerätedefinition gibt in dem Wert „`classes`“ ein Array von Referenzen auf *Geräteklassen* an. Diese Dokumente befinden sich in der Kollektion *DeviceClasses*.
Die referenzierten Dokumente definieren die *Geräteklasse*. Jedes dieser Dokumente enthält einen Namen für die *Geräteklasse*, sowie auch ein Array von Referenzen auf die Dokumente der *HighLevel* Kollektion. Die Dokumente der *Highlevel* Kollektion definieren eine Funktion, inklusive der benötigten Parameter. Diese Funktionen werden somit von einer *Geräteklasse* vorgeschrieben.

Im Folgenden sollen die einzelnen Schritte der *Pipeline* kurz erläutert werden. Dabei wird nicht erneut auf das Format der Operationen eingegangen.

Die *Pipeline* wird zunächst auf der *Device* Kollektion ausgeführt, da diese die Geräteklassen für ein bestimmtes Gerätemodell angibt. Die *Pipeline* nutzt somit ebenfalls das in Abbildung 5.7 auf Seite 52 gezeigte Dokument als Ausgangspunkt in der *Pipeline*.
Wie auch schon in der *Pipeline* für die Auslesung der Befehle, soll diese *Pipeline* nur das Dokument für ein bestimmtes Gerät ausgeben. Daher wird, erneut ein *$match* genutzt, welches die Dokumente anhand ihrer Felder „`manufacturer`“ und „`model`“ mit den angegebenen Werten vergleicht. Nach diesem Schritt ist nur noch das Dokument des betrachteten Gerätes in der *Pipeline* verbleibend.
Da es sich erneut um ein Array von Referenzen handelt, kann erneut die Operation *$lookup* genutzt werden, um diese Referenzen aufzulösen. In diesem Fall wird in die Kollektion *DeviceClasses* referenziert und es sind erneut die „`_id`“ Werte angegeben. Nach diesem Schritt besitzt das Dokument die in Abbildung 5.11 auf der nächsten Seite gezeigte Struktur.

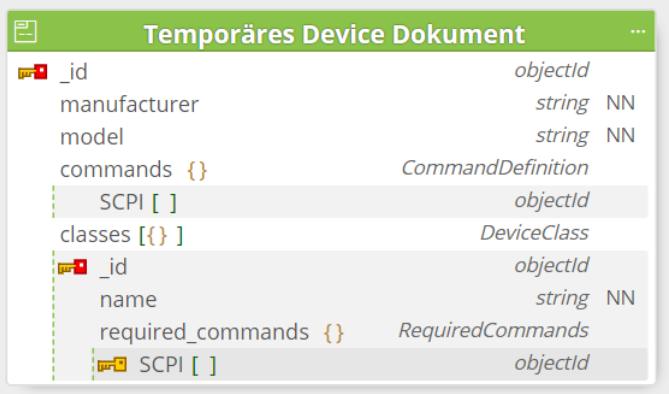

Abbildung 5.11.: *DeviceClasses-Pipeline* nach Auflösen der Referenzen in *DeviceClasses*

Da es sich um mehrere *DeviceClasses* handeln kann, werden die nun aufgelösten Referenzen erneut durch ein *$unwind* in einzelne Dokumente aufgeteilt.
Jede der Geräteklassen enthält ein Feld „`required_commands`“, welches ein eingebettetes Dokument ist. Dieses enthält im Wert „`SCPI`“ ein Array von Referenzen auf die *HighLevel*-Kollektion. Durch einen weiteren Aufruf von *$lookup* können nun auch diese Referenzen auf die *HighLevel* Kollektion aufgelöst werden. Das Ergebnis dieser Operation sind dann alle *HighLevel*-Funktionen, welche von der *Geräteklasse* vorgeschrieben werden.
Das Format der Dokumente nach dem Auflösen der Referenzen ist in Abbildung 5.12 auf der nächsten Seite dargestellt.

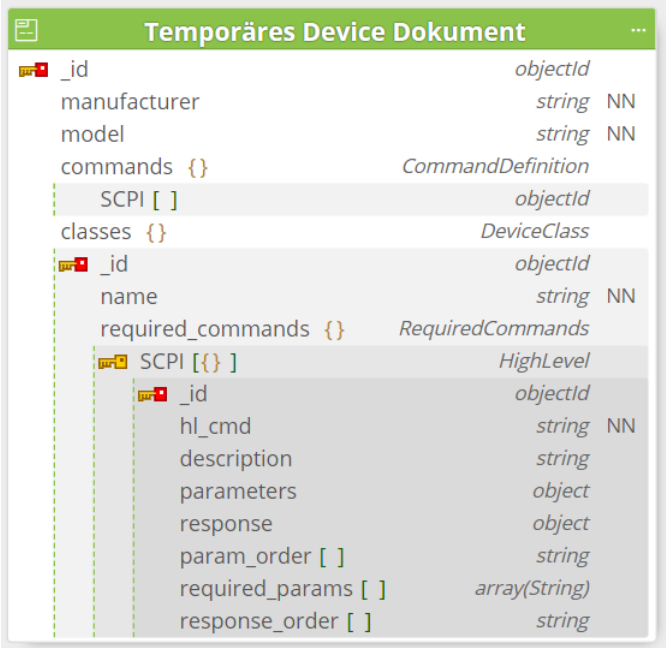

Abbildung 5.12.: *DeviceClasses-Pipeline* nach Auflösen aller Referenzen

Schließlich werden die Dokumente wieder in eins zusammengefasst. Wie auch in der anderen *Pipeline* enthalten hier alle Dokumente dieselbe „`_id`“, weshalb diese als Gruppierungskriterium genutzt werden kann. Die Werte „`model`“, „`manufacturer`“ und „`commands`“ sind in allen Dokumenten identisch, daher kann erneut der erste Wert aus diesen Dokumenten übernommen werden. Die einzelnen *Geräteklassen* werden durch ein *$push* als Gruppierungsvorschrift wieder in einem Array zusammengefügt.

Das Ergebnis ist ein Dokument, welches für das angegebene Gerät die *Geräteklassen* enthält. Jede dieser *Geräteklassen* enthält wiederum alle Funktionen, welche von der Klasse vorgeschrieben werden. Das Format ist noch einmal in Abbildung 5.13 auf der nächsten Seite dargestellt.

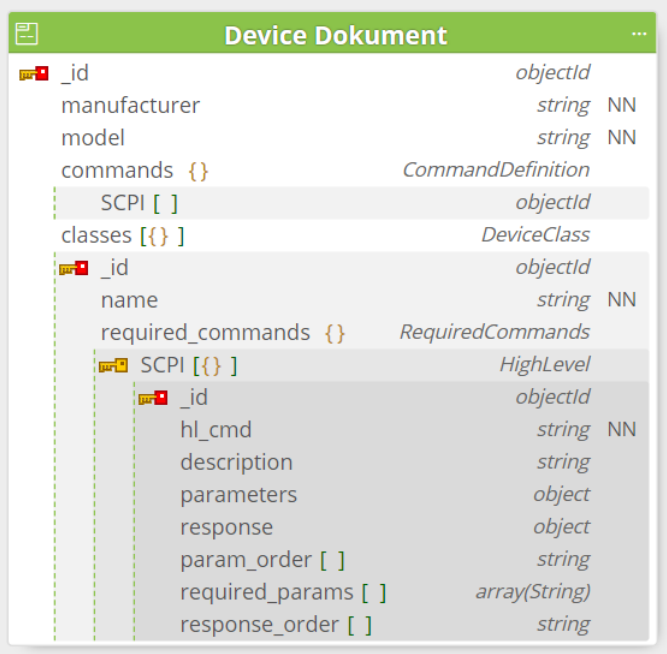

Abbildung 5.13.: *DeviceClasses-Pipeline* Ergebnis

Mit dem Ergebnis der *Pipeline* kann nun ein Gerät bezüglich seiner definierten Funktionen verglichen werden. Es wird daraufhin kontrolliert, dass ein Geräte alle Funktionen unterstützt, welche von seinen Geräteklassen als vorausgesetzt angesehen werden. Der Benutzer soll in der Lage sein, zwei Geräte einer *Geräteklasse* ohne Änderungen auszutauschen, falls die genutzten Funktionen durch diese *Geräteklasse* abgedeckt werden. Daher müssen alle Funktionen der Geräteklasse für ein Gerät umgesetzt sein. Falls bei der Überprüfung festgestellt wird, dass eine Funktion nicht umgesetzt wird, kann die korrekte Funktionsweise des Gerätes nicht sichergestellt werden. In diesem Fall, soll der Nutzer eine Warnung über die fehlenden Funktionen erhalten.

Eine vollständige Definition der *Pipeline* kann im Anhang A.2 gefunden werden.

6. Nutzung der Datenschnittstelle

Die Erstellung eines finalen Konzeptes für die Realisierung der Datenschnittstelle ist aktuell noch nicht möglich, da in den Laboren des ZEA-2 bisher keine einheitliche Vorgehensweise, wie zum Beispiel die genutzten Programmiersprachen, festgelegt wurde. Aktuell werden unter anderem die Programmiersprachen „MATLAB“, „Python“ und „C#“ genutzt. Aus diesem Grund werden in diesem Kapitel die generellen Ansätze für eine Datenschnittstelle erläutert.

Zunächst wird auf die Grundlagen der Nutzung eingegangen. Daraufhin werden die unterschiedlichen Varianten der Datenbeschaffung in Unterkapitel 6.2 und 6.3 behandelt. In Unterkapitel 6.4 werden anschließend die Rückgabewerte der Befehle betrachtet. Unterkapitel 6.5 befasst sich daraufhin mit der Kombination von einzelnen Befehlen und schließlich wird in Unterkapitel 6.6 eine mögliche Nutzung der Datenschnittstelle in *Python* dargestellt.

6.1. Generell

Die Nutzung der Datenschnittstelle soll mit allen Programmiersprachen möglich sein, welche die folgenden Kriterien erfüllen.

Die Programmiersprache

- unterstützt einen Treiber für *MongoDB*
- unterstützt die Kommunikation mit den Geräten über den `VISA`-Treiber, und
- bietet die nötigen Datentypen zur Darstellung der Werte

Wenn diese Bedingungen erfüllt sind, sollen zwei Möglichkeiten für die Nutzung der Schnittstelle möglich sein. Zum einen sollen die Daten aus der Datenbank zu Beginn des *Messskriptes* ausgelesen werden. Dies wird im Folgenden als *dynamisch* bezeichnet. Die zweite Variante soll ohne die Verbindung zur Datenbank auskommen. Dies wird als *statisch* bezeichnet.
Beide Varianten sollen unterstützt werden, damit die Daten aktuell aus der Datenbank genutzt werden können, aber auch eine Nutzung der *Messskripte* ohne eine Internet-

bzw. Intranetverbindung möglich ist.
Auf diese beiden Varianten soll in den folgenden Abschnitten eingegangen werden.

6.2. Dynamisch

Bei der *dynamischen* Nutzung der Datenschnittstelle sollen die Daten aus der Datenbank abgefragt werden. Zunächst wird das Gerät im *Messskript* über seinen Hersteller und seinen Modellnamen definiert.
Mit diesen Angaben soll kontrolliert werden, ob ein entsprechender Eintrag für dieses Gerätemodell existiert. Falls dies der Fall ist, wird die *Pipeline* für die Datenauslesung aus Unterkapitel 5.4 genutzt, um die Definitionen für die aktuellen Geräte auszulesen. Anhand dieser Daten können die definierten Funktionen des Gerätes genutzt werden. Damit eine Verbindung mit dem tatsächlichen Gerät hergestellt werden kann, müssen jedoch zusätzlich die Informationen für die Verbindung zum Gerät aus der Datenbank abgefragt werden. Dafür soll die Seriennummer des Gerätes angegeben werden. Diese ist für das gegebene Modell, von dem gegebenen Hersteller, eindeutig. Sie kann somit für die Abfrage der gerätespezifischen Verbindungsinformationen genutzt werden.
Ein Gerät kann dabei mehrere Verbindungsmethoden besitzen, wie zum Beispiel *TCP/IP* oder auch *USB*. In diesem Fall soll entweder eine Methode in dem *Messskript* angegeben werden, oder es sollen alle Methoden ausgetestet werden, bis eine Verbindung mit dem Gerät hergestellt werden konnte.

6.3. Statisch

Für eine *statische* Bereitstellung der Daten, welche somit nicht auf eine Verbindung mit der Datenbank angewiesen ist, gibt es mehrere Möglichkeiten für eine Umsetzung.

Eine Möglichkeit ist es, die Daten aus der Datenbank in einer lokalen Kopie zu speichern. Dafür werden die Ergebnisse der *Device-Pipeline* aus Unterkapitel 5.4 auf Seite 50 in lokalen *BSON*-Dateien gespeichert. Diese werden daraufhin wieder beim Start des *Messskriptes* eingelesen, um somit die Definitionen für die Gerätemodelle auch ohne die Datenbank verfügbar zu haben.
Dies hat jedoch den Nachteil, dass die lokalen Kopien nicht immer auf dem aktuellsten Stand, verglichen mit der Datenbank sind. Daher müssen diese Kopien regelmäßig erneuert werden.
Ein weiterer Nachteil ist, dass die Verbindungsoptionen nicht aus der Datenbank abgefragt werden können. Daher müssen in diesem Fall immer eine Verbindungsmethode,

inklusive der Verbindungsinformationen, wie zum Beispiel einer IP-Adresse angegeben werden.

Eine weitere Möglichkeit ist das Erstellen einer Bibliothek von den erstellten Geräten. In diesem Fall kann für jedes Gerät eine eigene Klasse erstellt werden, welche die verfügbaren Funktionen des Gerätes als Methoden bietet. Dies hat mehrere Vorteile. Zum einen ist die Unterstützung von Autovervollständigung in den Entwicklungsumgebungen, für die Erstellung der *Messskripte*, möglich. Außerdem können in diesem Fall die Datentypen bei *stark-typisierten* Programmiersprachen bereits in den Methoden eingetragen werden. Somit ist es unwahrscheinlicher, dass falsche Datentypen an die Funktionen übergeben werden.
Allerdings bringt die Bibliothek den Nachteil mit sich, dass sie nicht immer auf dem aktuellsten Stand ist und daher auch regelmäßig neu erstellt, und für die Messung auf die unterschiedlichen Computer verteilt werden muss. Außerdem muss diese Bibliothek für jede genutzte Programmiersprache individuell erstellt werden.
Um dies zu vereinfachen, kann eine *intermediäre Repräsentation* für die Programmiersprachen erstellt werden. Diese kann den Kontrollfluss der einzelnen Befehle abstrakt darstellen und daraufhin in die spezifischen Programmiersprachen überführen. Somit kann auch zukünftig der Aufwand für die Erweiterung der Datenschnittstelle reduziert werden, da nur die Generierung der *intermediäre Repräsentation* erweitert werden muss und die Überführung dieser in die Programmiersprachen identisch bleiben kann.

6.4. Aufruf und Rückgabe

Die entweder *dynamisch* oder *statisch* definierten Funktionen sollen für die Ausführung der Funktion genutzt werden. Dafür soll der Name der Funktion angegeben werden, gefolgt von den einzelnen Parametern für diese Funktion.
Nach diesem Aufruf können die Parameter auf mehrere Kriterien überprüft werden. Zunächst sollte überprüft werden, ob alle nötigen Parameter angegeben wurden und ob diese Kombination der Parameter gültig ist. Falls dies nicht der Fall ist, sollen die Parameter durch mögliche *Default*-Werte ersetzt werden, sofern diese bei der Definition angegeben wurden. Anschließend kann überprüft werden, ob die Parameter den richtigen Datentyp besitzen oder ob sie trivial in diesen konvertiert werden können. Trivial bedeutet hierbei, dass keine Informationen bei einer Konvertierung von einem Datentyp in den anderen verloren gehen. Dies ist zum Beispiel der Fall, wenn eine 8 Bit lange Ganzzahl angegeben wird, die Funktion aber eine 32 Bit Ganzzahl erwartet. In diesem Fall lässt sich der 8 Bit Wert ohne Informationsverlust in der 32 Bit Zahl darstellen.
Anschließend können die Parameter auf ihren Wertebereich kontrolliert werden, wenn dieser bei der Definition der Gerätefunktionen angegeben wurde. Sofern die Para-

meter gültig sind, können sie schließlich in die entsprechende *SCPI*-Repräsentation umgewandelt werden und in den *SCPI*-Befehl eingesetzt werden. Der Befehl kann daraufhin an das Gerät geschickt werden.
Für den Fall, dass die Parameter nicht vollständig angegeben wurden, einen falschen Datentypen besitzen, oder nicht im erlaubten Wertebereich liegen, soll ein Fehler ausgelöst werden. Dieser soll die Ausführung von Befehlen verhindern, die für das aktuelle Gerät nicht zulässig sind. Dadurch sollen mögliche Fehler im *Messskript* einfacher und früher erkannt werden. Es soll aber immer die Möglichkeit geben, diesen Fehler in eine Warnung abzuändern und den Befehl trotzdem zu versenden. Der Entwickler des *Messskriptes* hat somit die Möglichkeit, die gegebenen Einschränkungen zu deaktivieren. Somit können zum Beispiel die Funktionen kontrolliert werden, ob die Wertebereiche korrekt sind.
Der Entwickler soll außerdem die Möglichkeit haben, die *SCPI*-Befehle ausführen zu können, welche keine Funktion zugewiesen haben. In diesem Fall wird der Aufruf durch den *Programm Header*, statt durch den Funktionsnamen durchgeführt. Schließlich soll der Benutzer auch eigene Befehle eingeben können. Hierfür werden sie durch die Datenschnittstelle unverarbeitet an das Gerät weitergegeben. Dies soll unterstützt werden, damit auch Funktionen genutzt werden können, welche nicht in der Datenschnittstelle hinterlegt wurden. Diese Funktionen werden bei einem Austausch der Geräte vermutlich nicht mehr funktionieren.

Falls eine Rückgabe von einem Befehl geliefert wird, kann diese in unterschiedlichen Varianten geliefert werden. Dies ist vor allem von der genutzten Programmiersprache abhängig.
Die Rückgabe der Geräte wird als *Bytes* empfangen. Anhand der vorab definierten Formate der Rückgabewerte, sowie ihrer Datentypen, soll diese empfangene Nachricht in die entsprechenden Rückgabewerte und Datentypen überführt werden.
Der somit erstellte Rückgabewert soll die Daten sowohl über ihren Namen, als auch ihren Index, und somit ihrer Reihenfolge, zugreifbar machen. Dafür können in dynamischen Programmiersprachen zum Beispiel spezielle Datentypen zur Laufzeit erzeugt werden, welche die Rückgabedaten als Datenklasse bereitstellen. Andernfalls könnte zum Beispiel eine *Map* für den Zugriff auf die Daten möglich sein.
Es soll auch hier wieder möglich sein, dass der Benutzer eine eigene Abfrage definiert. In diesem Fall wird die *Query* unverarbeitet an die Geräte gesendet und die Antwort wird ebenfalls unverarbeitet als Bytes an den Nutzer zurückgegeben.

6.5. Kombinieren von Befehlen

Mehrere *SCPI*-Befehle können in einer Nachricht gesendet werden. Um dies zu ermöglichen, sollen Befehle „vorbereitet“ werden können. Mehrere dieser Befehle können hintereinander verkettet werden, bevor sie schließlich zum Gerät gesendet werden.

Dies ist relevant, da die Verkettung von machen Befehlen andere Ergebnisse liefern kann, als wenn sie individuell gesendet werden.
Falls mehrere Antworten aus den Befehlen erwartet werden, sollen diese den korrekten Befehlen zugewiesen werden. In diesem Falle werden die Rückgaben für alle Befehle zurückgegeben.

Falls einer der Befehle einen *Arbitrary ASCII*-Block oder *Arbitrary Program Data*-Block ohne spezifische Länge an das Gerät sendet, oder von diesem als Antwort erwartet, soll die Verkettung mit einem Fehler fehlschlagen. In diesem Fall ist das Senden eines Befehls nach einem solchen Block nicht mehr möglich, da das Ende des Parameters durch eine Beendigung der Übertragung dargestellt wird. Dasselbe ist bei den Antworten der Fall.

6.6. Beispiel der Nutzung

Die Nutzung der Datenschnittstelle soll in diesem Abschnitt an einem Beispiel dargestellt werden. Dieses Beispiel nutzt *Python* als Programmiersprache für die Erstellung des *Messskriptes*.

Es sollen zwei Geräte mit den Namen „Device1“ und „Device2“ existieren, welche jeweils vom *ZEA-2* hergestellt werden. Beide Geräte sollen eine Ausgabe per Text erlauben und „Device1“ kann zusätzlich die aktuelle Systemzeit ausgeben.
In den folgenden Abschnitten werden die Dokumente, welche in der Datenbank gespeichert werden, und deren Nutzung durch die Datenschnittstelle dargestellt.

Anmerkung: In den Dokumenten wird als „`_id`“ ein *String* verwendet, um das Beispiel zu vereinfachen. In der tatsächlichen Datenbank handelt es sich um einen Wert vom Typ *ObjectId*. Dies ist ein 24 Byte langer Wert, welcher *hexadezimal* als *String* gespeichert wird.

SCPI_Commands

Die Kollektion *SCPI_Commands* enthält die *SCPI*-Befehle der einzelnen Geräte. Dabei wird noch nicht festgelegt, welche Geräte diesen Befehl nutzen können. Die folgenden drei Dokumente sollen exemplarisch die genannten Funktionalitäten umsetzen.

```
1  {
2      "_id"         : "SCPI_SET_DISP_TEXT",
3      "command"     : ":SET:DISPLAY:TEXT <text>",
4      "parameters" : {
5          "text": {
6              "description": "Der Text für das Display",
7              "datatype"    : "string",
8          }
9      },
10     "required_params": [["text"]]
11 }
```

Dokument 6.1.: setDisplay *SCPI* Definition für Gerät 1

Dokument 6.1 definiert den Befehl „`:SET:DISPLAY:TEXT`“ in Zeile 3, welcher den Parameter „`text`“ annimmt. Dieser ist in Zeile 5 als Parameter für diesen Befehl angegeben und in Zeile 7 als ein *String* definiert. Der Parameter soll dabei immer notwendig sein, weshalb er in Zeile 10 als benötigter Parameter angegeben ist.

```
1  {
2      "_id"         : "SCPI_DISP_TEXT",
3      "command"     : ":DISP:TEXT <data>",
4      "parameters" : {
5          "data": {
6              "description": "Text zum Anzeigen auf dem Display",
7              "datatype"    : "string",
8          }
9      },
10     "required_params": [["data"]]
11 }
```

Dokument 6.2.: setDisplay *SCPI* Definition für Gerät 2

Ähnlich wie im vorherigen Dokument wird in Dokument 6.2 ein Befehl zur Ausgabe des Textes definiert. In diesem Fall heißt dieser jedoch „`:DISP:TEXT`“ und akzeptiert den Parameter „`data`“ anstatt „`text`“.

```
1  {
2      "_id"      : "SCPI_SYSTEM_TIME?",
3      "command" : ":SYSTem:TIME?"
4  }
```

Dokument 6.3.: getTime *SCPI* Definition

Dokument 6.3 definiert die *Query* für die Ausgabe der Systemzeit. Da keine Parameter erwartet werden, ist nur das „`command`“ Feld, in Zeile 3, relevant für die Definition.

HighLevel

Die *HighLevel* Kollektion beinhaltet die definierten Funktionen der Geräte. In diesem Beispiel werden zwei Funktionen genutzt. Einmal das Ausgeben von Text auf einem Display und das Auslesen der Systemzeit. Es wird in dieser Kollektion noch nicht festgelegt, welche Geräte welche *HighLevel* Funktionen unterstützen.
Die folgenden zwei Dokumente stellen die Definitionen in der *HighLevel* Kollektion dar.

```
1  {
2      "_id"             : "HL_setDisplayText",
3      "description"     : "Gibt den gegeben Text auf dem Display aus",
4      "hl_cmd"          : "setDisplayText",
5      "parameters"      : {
6          "text" : {
7              "description" : "Der Text der auf dem Display gezeigt wird",
8              "datatype"    : "string"
9          }
10     },
11     "required_params" : ["text"],
12     "param_order"     : ["text"]
13 }
```

Dokument 6.4.: setDisplay *HighLevel* Definition

Dokument 6.4 definiert die Funktion „setDisplayText“ in Zeile 4. Es wird zusätzlich in Zeile 6 definiert, dass der Parameter „`text`“ erwartet wird. Dieser muss ebenfalls ein *String* sein (Zeile 8). Er muss außerdem immer angegeben werden (Zeile 11) und ist der erste Parameter im Aufruf (Zeile 12).

```
1  {
2      "_id"            : "HL_getTime",
3      "description"    : "Ließt die aktuelle Systemzeit aus",
4      "hl_cmd"         : "getSystemTime",
5      "response"       : {
6          "hours"   : {"datatype" : "uint8"},
7          "minutes" : {"datatype" : "uint8"},
8          "seconds" : {"datatype" : "single"}
9      },
10     "response_order" : ["hours", "minutes", "seconds"]
11 }
```

Dokument 6.5.: getSystemTime *HighLevel* Definition

Das Dokument 6.5 definiert die Abfrage der Systemzeit durch die Funktion „getSystemTime“. Es werden drei Rückgabewerte in den Zeilen 5 bis 9 definiert. Zunächst die Werte *hours* und *minutes*, welche mindestens eine 8 Bit positive Ganzzahl benötigen (Zeile 6 und 7). Zeile 8 definiert schließlich noch den Wert *seconds*, welcher mindestens einen *float* („single precision“ Datentyp nach *IEEE754*) benötigt. Wenn die Ergebnisse iteriert werden sollten, ist die Reihenfolge ebenfalls auf *hours*, *minutes*, *seconds* in Zeile 10 festgelegt.

Mit diesen Dokumenten sind nun die möglichen Funktionen und somit das Interface für den Nutzer definiert.

SCPI_Command_Definitions

Die Kollektion *SCPI_Command_Definitions* muss nun die *HighLevel* Funktionen mit den *SCPI*-Befehlen verbinden. Es wird dabei noch nicht angegeben, welches Gerät diese Kombination von *HighLevel* und *SCPI*-Befehl nutzt, sondern nur welche Kombinationen der Befehle möglich sind.
Die folgenden Beispieldokumente zeigen eine mögliche Kombination für diese Verbindung.

```
1  {
2      "_id"    : "Device1_setDisplayText",
3      "command": "SCPI_SET_DISP_TEXT",
4      "hl"     : "setDisplayText",
5  }
```

Dokument 6.6.: Gerätespezifische setDisplay Definition für „Device1“

Das Dokument 6.6 definiert die Verbindung des *SCPI*-Befehls und der *HighLevel*-Funktion aus den Dokumenten 6.1 und 6.4. Dafür werden in den Feldern „`command`“ (Zeile 2) und „`hl`“ (Zeile 3) die Werte der „`_id`“ Felder der beiden Dokumente angegeben. Da beide Befehle bereits die gleichen Namen für die Parameter nutzen und keine Rückgabe von dem Befehl erwartet wird, müssen keine weiteren Einstellungen vorgenommen werden.

```
1 {
2     "_id"    : "Device2_setDisplayText",
3     "command": "SCPI_DISP_TEXT",
4     "hl"     : "HL_setDisplayText",
5     "mapping": {
6         "data": "text"
7     }
8 }
```

Dokument 6.7.: Gerätespezifische setDisplay Definition für „Device2“

Das Dokument 6.7 definiert die Verbindung der beiden Befehl aus den Dokumenten 6.2 und 6.4. Die *HighLevel*-Funktion nutzt jedoch den Parameternamen „`text`“, während der *SCPI*-Befehl den Parameter „`data`“ erwartet. Daher wird in Zeile 6 der Wert von „data“ an den Parameter „`text`“ gebunden.

```
1 {
2     "_id"             : "Device1_getSystemTime",
3     "command"         : "SCPI_SYSTEM_TIME?",
4     "hl"              : "HL_getTime",
5     "response"        : "<hour>,<minute>,<second>",
6     "response_types"  : {
7         "hour"  : {"datatype": "uint8"},
8         "minute": {"datatype": "uint8"},
9         "second": {"datatype": "single"}
10    },
11    "response_mapping": {
12        "minutes": "minute",
13        "seconds": "second",
14        "hours"  : "hour"
15    }
16 }
```

Dokument 6.8.: Gerätespezifische getTime Definition für „Device1“

Das Dokument 6.8 definiert die Verbindung des *SCPI*-Befehls und der *HighLevel*-Funktion aus den Dokumenten 6.3 und 6.5. In diesem Dokument werden ebenfalls die Rückgaben des Gerätes definiert. Dafür definiert Zeile 5 das Format der Rückgabe. In

diesem Fall wird zunächst der Wert *hour* erwartet, daraufhin *minute* und schließlich *second*. Alle drei Rückgabewerte sind dabei durch Kommata getrennt.
Die Zeilen 7 und 8 definieren zusätzlich, dass es sich bei den Werten *hour* und *minute* jeweils um eine 8 Bit Ganzzahl ohne Vorzeichen handelt. Die Zeile 9 wiederum definiert, dass *second* ein Parameter mit dem Datentyp einer *single Precision* Zahl hat. Schließlich müssen die Rückgaben noch den Rückgaben in der *HighLevel*-Funktion zugewiesen werden. Die Funktion ist in diesem Fall *getSystemTime*, da der „`hl`“ Wert in Zeile 4 auf das Dokument 6.5 verweist. In diesem sind die Rückgabewerte als *hours*, *minutes* und *seconds* definiert. Da diese Namen nicht mit denen aus der „`response`“ in Zeile 5 übereinstimmen, werden die Werte neu zugewiesen. Zeile 12 legt hierbei fest, dass der Rückgabewert der Funktion *minutes* an den Wert *minute* gebunden wird. Zeile 13 wiederholt dies für die Werte *seconds* und *second*, sowie Zeile 14 für *hours* und *hour*.

Mit diesen Dokumenten wurden nun die Verbindungen zwischen den *SCPI*-Befehlen, sowie den *HighLevel*-Funktionen hergestellt. Damit diese Kombinationen nun von den Geräten genutzt werden können, muss den Geräten noch zugewiesen werden, welche von den Kombinationen sie nutzen sollen.

Devices

Der letzte für dieses Beispiel relevante Schritt ist die *Devices* Kollektion. Diese enthält die finale Definition der beiden Geräte, sowie welche Befehle die Geräte jeweils umsetzen.

Die möglichen Dokumente für die Definition der beiden Geräte werden im Folgenden vorgestellt.

```
1  {
2      "model"        : "Device1",
3      "manufacturer" : "ZEA-2",
4      "commands"     : {
5          "SCPI" : [
6              "Device1_setDisplayText",
7              "Device1_getSystemTime"
8          ]
9      }
10 }
```

Dokument 6.9.: Definition für „Device1“

Das Dokument 6.9 definiert das „Device1“. Es wird dabei in Zeile 2 festgelegt, dass

das Gerät den Modellnamen „Device1“ erhält und der Hersteller wird in Zeile 3 als „ZEA-2“ festgelegt.
In Zeile 5 wird zusätzlich festgelegt, dass das Gerät *SCPI*-Befehle unterstützt. Die Zeilen 6 und 7 verweisen dabei auf die Dokumente 6.6 und 6.8. Durch diese Referenzen werden einerseits die *HighLevel*-Funktionen für das Gerät verfügbar gemacht und andererseits festgelegt, welche *SCPI*-Befehle diese Funktion für das gegebene Gerät umsetzen.

```
1 {
2     "model"        : "Device2",
3     "manufacturer" : "ZEA-2",
4     "commands"     : {
5         "SCPI" : [
6             "Device2_setDisplayText"
7         ]
8     }
9 }
```

Dokument 6.10.: Definition für „Device2“

Das Dokument 6.10 definiert ebenfalls ein Gerät. In diesem Falle wird es jedoch „Device2“ genannt und es unterstützt andere Befehle als das „Device1“. Der Grund dafür ist, dass in Zeile 6 auf ein anderes Dokument verwiesen wird, welches in diesem Fall Dokument 6.7 ist. Da dieses einen anderen *SCPI*-Befehl der Funktion „setDisplayText“ zugewiesen hat, wird bei der Ausführung der Funktion ein anderer *SCPI*-Befehl genutzt, als bei „Device1“. Da außerdem keine Referenz auf ein Objekt existiert, welches die „getSystemTime“ Funktion enthält, ist diese Funktion für das „Device2“ nicht verfügbar.

Die Erstellung dieser Dokumente soll ein einmaliger Prozess sein und außerdem nicht von jedem Labormitarbeitenden durchgeführt werden müssen.
Das Hinzufügen neuer Dokumente ist nur nötig, wenn ein neues Gerätemodell für die Nutzung in der Datenschnittstelle hinzugefügt werden soll.
Außerdem soll die Erstellung dieser Dokumente zukünftig durch die Verwaltungssoftware der Datenschnittstelle ermöglicht werden. Diese soll die graphische Erstellung der Dokumente über eine Benutzeroberfläche erlauben und das Format der Daten automatisch aus den Eingaben des Benutzers generieren.
Außerdem bieten unterschiedliche Hersteller die Geräteinformationen zum Beispiel als *XML*-Datei an, welche von der Verwaltungssoftware ausgewertet und automatisch in die Datenschnittstelle integriert werden können.

Messskript

Dieser Abschnitt beschreibt eine mögliche Nutzung der Datenschnittstelle in der Programmiersprache *Python* und soll exemplarisch das Vorgehen eines Nutzers bei der Erstellung eines *Messskriptes* erläutern. Dies ist somit das Vorgehen, welches für den Benutzer relevant ist.

Der folgende Ausschnitt eines möglichen *Messskriptes* in Abbildung 6.11 soll die Nutzung der beiden zuvor definierten Geräte demonstrieren.

```
1  # Dynamische Nutzung
2  di = DataInterface(host="ZEA2_Measurements_Server")
3
4  # Angabe der Seriennummer
5  dev_1 = di.get_scpi_device(model="Device1", manufacturer="ZEA-2",
6                  serial="00000001", connection=ConnectionTypes.TCP)
7  # Explizite Angabe der Verbindung
8  dev_2 = di.get_scpi_device(model="Device2", manufacturer="ZEA-2",
9                  connection=ConnectionTypes.TCP, ip="192.168.1.1")
10
11 dev_1.getCommand('setDisplayText', text='Text fuer Geraet 1').send()
12 # Sendet an Gerät 1: `:SET:DISPLAY:TEXT "Text fuer Geraet 1"´
13 dev_2.getCommand("setDisplayText", "Geraet 2 Text").send()
14 # Sendet an Gerät 2: `:DISP:TEXT "Geraet 2 Text"´
15
16 # Angenommen die aktuelle Uhrzeit wäre 10:15:25 Uhr und 120 Millisekunden
17 current_time = dev_1.getCommand("getSystemTime").send()
18 # Sendet an Gerät 1: `:SYSTem:TIME?´
19 # Liefert als Ergebnis: `getSystemTimeResult(hours=10, minutes=15,
   ↪  seconds=25.12)´
20
21 combined = dev_1.getCommand('setDisplayText', text='Kombinierter Befehl') |
   ↪  dev_1.getCommand("getSystemTime")
22 # Kombiniert die beiden Befehle in einen zusammengesetzten
23
24 result = combined.send()
25 # Sendet an Gerät 1:
26 #   `:SET:DISPLAY:TEXT "Kombinierter Befehl";:SYSTem:TIME?´
27 # result entspricht danach:
28 #   [None, getSystemTimeResult(hours=10, minutes=15, seconds=25.12)]
```

Programmcode 6.11.: Beispiel einer Nutzung der Datenschnittstelle in Python

Zunächst wird in Zeile 2 die Datenschnittstelle erstellt, welcher als „`host`“ der Name des Datenbankservers übergeben wird. In diesem Beispiel ist der Datenbankserver

über den Namen „ZEA2_Measurement_Server“ erreichbar. Für eine *statische* Nutzung kann zum Beispiel ein Pfad zu einem Ordner angegeben werden, welcher die unterschiedlichen *BSON*-Dateien für die Definition der Geräte enthält.

In Zeile 5f wird nun das erste Gerät aus der Datenbank ausgelesen. Dieses ist vom Modell „Device1“, welches vom ZEA-2 hergestellt wird. Der Wert „`serial`“ gibt in diesem Fall an, welches tatsächliche Gerät von diesem Modell genutzt wird. Das Speichern dieser Verbindungsinformationen ist jedoch nicht Teil dieser Arbeit und wird daher nicht weiter betrachtet. Es wird davon ausgegangen, dass die Verbindungsdaten für jedes Gerät vorhanden sind. Ebenso wird davon ausgegangen, dass einer Verbindung mit den Geräten möglich ist. Der Wert „`connection`“ wurde dazu angeben, dass die Verbindung über *TCP/IP* aufgebaut werden soll.

Zeile 8f lädt die Definition für das „Device2“ aus der Datenbank. In diesem Falle wird keine Seriennummer angegeben, weshalb die IP-Adresse des Gerätes explizit angegeben werden muss. Sie kann nicht wie bei „Device1“ aus der Datenbank abgefragt werden.

Die Objekte „`dev_1`“ und „`dev_2`“ repräsentieren die geladenen Geräte. Diese werden für die Kommunikation mit dem jeweiligen Gerät genutzt und auf ihnen werden die definierten Funktionen aufgerufen. Eine dieser Funktionen soll „`getCommand`“ sein.
Die Funktion „`getCommand`“ soll den Funktionsnamen der auszuführenden Funktion als ersten Parameter erhalten. Alle weiteren Parameter werden als Parameter für den *SCPI*-Befehl bzw. die *HighLevel*-Funktion angesehen. Das Ergebnis des „`getCommand`“ Aufrufs ist ein vorbereiteter Befehl. Dieser beinhaltet bereits die nötigen Daten, damit er an das Gerät gesendet werden kann. Dies kann durch den Aufruf der Methode „`send()`“ erreicht werden. Er wird daraufhin an das Gerät gesendet und mögliche Antworten des Gerätes werden abgewartet. Diese Antworten werden daraufhin anhand der definierten Datentypen und Formate für Rückgabewerte ausgewertet und sind das Ergebnis der „`send()`“ Methode.

In Zeile 11 wird dies an der Funktion „setDisplayText“ verdeutlicht. Diese Funktion wird mit dem Objekt „`dev_1`“ erstellt, wodurch die Definitionen für „Device1“ genutzt werden.
Für den Aufruf der Funktion wird der Parameter „`text`“ angegeben. Dieser wird als Text an das Gerät gesendet, um diesen auf dem Gerät auszugeben. Dafür wird der Parameter in den *SCPI*-Befehl eingefügt, welcher für die Funktion „setDisplayText“ in Dokument 6.6 hinterlegt wurde. In diesem Falle war dies „`:SET:DISPLAY:TEXT <text>`“. Da das „`send()`“ unmittelbar aufgerufen wurde, wird der Befehl sofort an das Gerät versendet. Das Ergebnis des Aufrufs ist ein *None*, da keine Rückgabe des Gerätes für diesen Befehl erwartet wurde.

Die Zeile 13 führt dieselbe Funktion „setDisplayText“ für „Device2“ aus. In diesem Fall wird der Befehl „`:DISP:TEXT <data>`“ ausgeführt, welcher für „Device2“ in Do-

kument 6.7 hinterlegt wurde. Der erste Parameter war dabei als „text“ angegeben, weshalb der Wert „Geraet 2 Text“ an den Parameter „`text`“ gebunden wird. Während der Auswertung der Funktion wird dieser für das Einfügen in den *SCPI*-Befehl in „`data`“ umbenannt.

Zeile 17 zeigt einen möglichen Aufruf einer *Query*. Für die Funktion „getSystemTime“ war eine Rückgabe in Dokument 6.8 definiert worden. Daher wird in diesem Fall ein Objekt vom Typ „`getSystemTimeResult`“ definiert, welches die angegebenen Rückgabewerte mit ihren Namen und in der angegebenen Reihenfolge definiert. Es wird die aktuelle Zeit des Geräts in der Reihenfolge Stunden, Minuten und Sekunden angegeben. Durch zum Beispiel die Angabe „`current_time.hours`“ soll auf den Rückgabewert „`hours`“ zugegriffen werden können. Diese Werte können im weiteren Verlauf des *Messskripts* genutzt werden.

Die Ausführung von mehreren *SCPI*-Befehlen innerhalb einer Nachricht kann ein anderes Ergebnis liefern, als wenn diese Befehle einzeln ausgeführt werden. Daher sollen die Befehle kombiniert werden können. Dies ist in Zeile 21 einmal dargestellt. In dieser werden die Befehle „setDisplayText“ und „getSystemTime“ jeweils vorbereitet, allerdings noch nicht versendet. Durch den „`|`“ Operator werden diese Befehle in einen einzelnen kombiniert. Dabei muss darauf geachtet werden, dass der Befehl auch zum gleichen Gerät gehört, da sonst eine Verkettung keine korrekten Befehle erzeugen würde.
In Zeile 24 wird der zuvor kombinierte Befehl abgeschickt. Der „setDisplayText“ Befehl gibt keine Rückgaben zurück, während der „getSystemTime“ Befehl die aktuelle Systemzeit ausgibt. Da zwei Befehle kombiniert wurden, werden auch zwei Rückgaben von dem Befehl geliefert. Der erste Rückgabewert ist dabei ein *None* und gehört zu dem ersten Befehl. Dies war hier der „setDisplayText“ Befehl, welcher keine Rückgaben liefert. Der zweite Wert der Rückgaben ist das Ergebnis des zweiten Befehls, welcher hier das Auslesen der Systemzeit war.
Dieses Vorgehen der Verkettung und Rückgabe mehrerer Rückgabewerte soll für beliebig viele Befehle in Folge möglich sein.

Abschließend ist zu dem gezeigten Beispiel in Codeausschnitt 6.11 zu sagen, dass die Datenschnittstelle sich möglichst einfach in die Programmiersprache integrieren lassen soll. Daher soll die Datenschnittstelle möglichst stark an die Fähigkeiten und den generellen Aufbau der Programmiersprache angepasst werden, sodass die Nutzung relativ intuitiv möglich ist.
Die tatsächliche Umsetzung und Nutzung der Datenschnittstelle ist somit stark von der Programmiersprache abhängig und es kann kein allgemeines Format aller *Messskripte* bzw. aller Datenschnittstellen für jede Programmiersprache definiert werden.

7. Zusammenfassung und Ausblick

Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Konzept für eine universelle Datenschnittstelle erstellt. Diese soll die Kommunikation aus einem *Messskript* mit beliebigen Messgeräten durch das *SCPI*-Befehlsformat erlauben. Durch diese Schnittstelle soll das Erstellen der *Messskripte*, in unterschiedlichen Programmiersprachen, vereinfacht werden, ebenso wie die Wartung der Skripte.

SCPI ist ein von vielen programmierbaren Geräten unterstütztes Befehlsformat, welches *ASCII Strings* für die Kommunikation mit dem Kontrollsystem nutzt. Es definiert ein Schema für die Erstellung von Befehlen und Abfragen. Unterschiedliche Geräte unterstützen jedoch andere Befehle für die gleichen Operationen, wie zum Beispiel dem Einschalten einer Spannungsquelle oder dem Auslesen eines Messwertes. Diese Funktionen sollen durch die Datenschnittstelle verallgemeinert werden.

Die Datenschnittstelle ermöglicht es, generelle Funktionen, wie das Setzen und Auslesen einer Spannung, geräteübergreifend zu definieren. Für ein bestimmtes Gerätemodell werden die *SCPI*-Befehle hinterlegt, welche diese Funktion realisieren. Dabei ist es ebenfalls möglich, die nötigen Parameter für die Befehle anzugeben.

Die Datenschnittstelle nutzt eine *MongoDB* Datenbank zur Speicherung der Gerätemodelle. Diese speichert in insgesamt fünf Kollektionen die nötigen Informationen über die Geräte, deren Funktionen, die *SCPI*-Befehle und deren Parameter. Diese Kollektionen werden durch *Schema-Validatoren* auf die korrekte Formatierung der Daten überprüft und durch eine dafür erstelle externe Software auf die inhaltliche Korrektheit kontrolliert.

Durch die Erstellung von *Geräteklassen* können Funktionen definiert werden, welche von allen Geräten der *Geräteklasse* unterstützt werden müssen. Somit können Geräte einer *Geräteklasse* ausgetauscht werden, ohne Änderungen am Ablauf der *Messskripte* vornehmen zu müssen. Dies ermöglicht einen einfachen Austausch der Messgeräte, falls diese Defekt werden, oder ein anderes Modell benötigt wird.

In den *Messskripten* sollen die Gerätedaten entweder zu Beginn aus der Datenbank abgefragt werden, oder die Daten der Datenbank sollen in eine lokale Bibliothek überführt werden. Diese soll für unterschiedliche Programmiersprachen erstellt werden können. Dafür ist nötig, dass die Kommunikation über den `VISA` Treiber mit den Geräten möglich ist.

Zu Beginn des *Messskripts* werden die genutzten Geräte angegeben und es soll über einen „DeviceHandle“ die Kommunikation mit dem Gerät erlaubt werden. Durch Angabe der Funktionsnamen werden die nötigen Informationen aus der Gerätedefinition ausgelesen. Die nötigen Parameter für die Ausführung der Funktion werden zusätzlich angegeben. Diese werden zunächst auf ihren Datentypen kontrolliert und gegebenenfalls noch auf ihre möglichen Werte oder Wertebereiche. Schließlich werden die Parameter in den für das Gerät hinterlegten *SCPI*-Befehl integriert und können an dieses versendet werden. Falls eine Rückgabe von dem Gerät erwartet wird, wird diese eingelesen. Anhand des definierten Rückgabeformates und der angegebenen Datentypen wird diese Antwort automatisch in die einzelnen Werte zerlegt und in die entsprechenden Datentypen für die genutzte Programmiersprache überführt. Somit können die gelesenen Werte in den *Messskripten* weiter verarbeitet oder gespeichert werden.

Die Datenschnittstelle ist dabei auf eine möglichst intuitive Nutzung aus den genutzten Programmiersprachen ausgelegt. Sie soll somit die Erstellung der *Messskripte* vereinfachen, indem der Aufwand für das Eintragen der *SCPI*-Befehle auf eine einzige Definition in der Datenbank reduziert wird. Somit können die Geräte über ihre geräteübergreifenden Funktionen in den *Messskripten* genutzt werden und der Austausch der Geräte wird durch eine einzelne Änderung in der Initialisierung des *Messskriptes* möglich.

Ausblick

Die in dieser Arbeit konzipierte Datenschnittstelle wird im weiteren Verlauf der Entwicklung eines *Laborsetups* für das ZEA-2 realisiert und erweitert.

Zunächst ist eine Realisierung der Datenschnittstelle in ausgewählten Programmiersprachen vorgesehen. Diese sollen daraufhin in den *Messskripten* getestet werden. In Zusammenarbeit mit den Labormitarbeitern wird daraufhin das weitere Vorgehen für einen ZEA-2 weiten Einsatz der Datenschnittstelle geplant. Dies wird unter anderem die genutzten Programmiersprachen, Berechtigungen für unterschiedliche Nutzer oder Nutzergruppen, oder auch zusätzliche Übertragungsprotokolle umfassen.
Zum Beispiel ist die Erweiterung der Datenschnittstelle um *JTAG*-Kommunikation für die Ansteuerung von, im ZEA-2 entwickelten, integrierten Schaltkreisen eine mögliche Erweiterung.
Es wurde ebenfalls schon in Betracht gezogen, nicht nur einzelne *SCPI*-Befehle in der Datenbank zu speichern, sondern ganze „Prozeduren“. Diese können zum Beispiel das Einstellen einer Temperatur an einem Kälteschrank umfassen. Dabei würde in regelmäßigen Abständen die Temperatur ausgelesen und anhand dieser die Einstellungen des Kälteschrankes angepasst, bis die gewünschte Temperatur über einen bestimmten Zeitraum konstant beibehalten wird.

Während der Konzeptionierung der Datenbank, ihrer Kollektionen, und ihrer *Schema-Validatoren*, wurde der Aufbau bereits anhand von einzelnen „Testdaten“ und ausgewählten Gerätefunktionen getestet. Diese Datenbank soll nun mit realen Daten für die Gerätemodelle gefüllt werden, damit diese schließlich in der Datenschnittstelle genutzt werden können.

Eine weitere Erweiterung ist die Erstellung einer Verwaltungssoftware für die Datenschnittstelle. Über diese sollen neue Geräte leichter in die Datenbank hinzugefügt werden. Diese Verwaltungssoftware kann in die von Klara Schnorrenberg konzeptionierte *Datenmanagementsoftware* integriert werden. Der aktuelle Plan für das *Laborsetup* wurde in Abbildung 3.1 auf Seite 18 bereits vorgestellt.
Ebenso muss eine externe Software für die Validierung der Daten in der Datenschnittstelle erstellt werden. Der Grund dafür ist, dass nicht alle Fehler in den Daten durch die *Schema-Validatoren* verhindert werden können.

Es wird ebenfalls untersucht, ob eine Generierung einer „Bibliothek“ für die Nutzung der einzelnen Gerätemodelle eine sinnvolle Erweiterung für die Nutzung der *Messskripte* darstellt. Für diese wird die Entwicklung einer *intermediären Repräsentation* für die Definition der Programmabläufe genauer betrachtet. Somit kann die Generierung dieser Repräsentation erweitert werden, wenn neue Funktionen zu der Datenschnittstelle hinzugefügt werden. Die Repräsentation kann dann in die spezifischen Strukturen der Programmiersprachen für die Kommunikation überführt werden.

Schließlich kann auch die Nutzung des `VISA` Treibers weiter untersucht werden. Dieser ist gegebenenfalls nicht für alle Kommunikationen mit den Geräten geeignet. Daher kann eine Umsetzung einer Gerätekommunikation ohne den `VISA` Treiber in Betracht gezogen werden.

A. Pipelines

In diesem Anhang werden die in Unterkapitel 4.2 auf Seite 23 beschriebenen *Pipelines* dargestellt.

Eine *Pipeline* kann auf die Felder des aktuellen Dokumentes zugreifen. Dafür ist es nötig, ein „$“ vor den Namen des Feldes zu schreiben. Werte mit einem „$$“ sind beim Start der Pipeline definiert und als Konstante während der Ausführung anzusehen.

Die *Pipelines* sind in *MongoDB Shell Syntax* angegeben. Es werden zusätzliche Kommentare durch ein „//“ hinzugefügt. Diese sind nicht Teil der Pipeline.

Pipeline A.1: Auslesen der Gerätefunktionen

```
[
    {
    // Die Werte "model" und "manufacturer" werden
    // beim Aufruf der Pipeline angegeben.
    // Die Auswertung dieser Werte muss daher in
    // einem "$expr" Block geschehen.
    // Sie werden durch "$$model" und "$$manufacturer"
    // mit den Werten in dem Dokument verglichen
    "$match": {
      '$expr': {
        '$and': [
          {'$eq': ['$model', '$$model']},
          {"$eq": ["$manufacturer", "$$manufacturer"]}
        ]
      }
    }
    },

    $lookup: {
        // Löst die Verweise aus dem Array
        // "commands.SCPI" anhand ihrer "_id"
        // in die Dokumente aus "SCPI_Command_Definitions"
        // auf und kopierte diese in das Array zurück.
        from: 'SCPI_Command_Definitions',
```

```
25          localField: 'commands.SCPI',
26          foreignField: '_id',
27          as: 'commands.SCPI'
28      },
29
30    {
31      // Entpackt die zuvor aufgelösten Referenzen
32      '$unwind': {
33        'path': '$commands.SCPI',
34        'preserveNullAndEmptyArrays': true
35      }
36    },
37
38    {
39      // Löst den Verweis der HighLevel Funktion auf
40      '$lookup': {
41        'from': 'HighLevel',
42        'localField': 'commands.SCPI.hl',
43        'foreignField': '_id',
44        'as': 'commands.SCPI.hl'
45      }
46    },
47
48    {
49      // Wandelt das Ergebnisarray aus dem lookup
50      // in ein einzelnes Dokument um
51      '$unwind': {
52        'path': '$commands.SCPI.hl',
53        'preserveNullAndEmptyArrays': true
54      }
55    },
56
57    {
58      // Löst den Verweis des SCPI-Befehls auf
59      '$lookup': {
60        'from': 'SCPI_Commands',
61        'localField': 'commands.SCPI.command',
62        'foreignField': '_id',
63        'as': 'commands.SCPI.command'
64      }
65    },
66
67    {
68      // Entpackt das Ergebnissarray
69      '$unwind': {
70        'path': '$commands.SCPI.command',
```

```
71        'preserveNullAndEmptyArrays': true
72      }
73    },
74
75    {
76      // Gruppiert die Dokumente anhand ihrer _id zu einem
77      '$group': {
78        '_id': '$_id',
79
80        'model': {
81        // $first meint, dass der Wert des ersten betrachteten
82        // Dokumentes genutzt wird. Alle Weiteren werden
             verworfen
83          '$first': '$model'
84        },
85
86        'manufacturer': {'$first': '$manufacturer'},
87        'classes': {'$first': '$classes'},
88
89        'SCPI_CMD': {
90        // $push fügt die Werte alle Dokumente in ein Array
             zusammen
91        // Hier muss der temporäre Wert "SCPI_CMD" genutzt
             werden
92          '$push': '$commands.SCPI'
93        }
94      }
95    },
96
97    {
98      // Speichert den temporären Wert "SCPI_CMD" zurück
99      // in das Dokument "commands"
100     '$addFields': {
101       'commands.SCPI': '$SCPI_CMD'
102     }
103   },
104
105   {
106     // Entfernt die temporäre Variable "SCPI_CMD"
107     '$project': {
108       'SCPI_CMD': 0
109     }
110   }
111
112   // Ende der Pipeline
113 ]
```

Pipeline A.2: Auslesen der Vorgaben durch die Geräteklassen eines Gerätes

```
1
2  [
3      {
4       // Filtert die entsprechenden Dokumente anhand
5       // des angegebenen Modells und des Herstellers
6       $match: {
7        $expr: {
8         $and: [
9          {$eq: ['$model','$$model']},
10         {$eq: ['$manufacturer','$$manufacturer']}
11        ]
12       }
13      }
14     },
15
16     {
17      // Löst alle Referenzen aus dem Array "classes" auf
18      // und ersetzt diese durch die Dokumente aus
19      // DeviceClasses mit der selben "_id"
20      $lookup: {
21         from: 'DeviceClasses',
22         localField: 'classes',
23         foreignField: '_id',
24         as: 'classes'
25     }
26
27     {
28      // Entpackt das Array der "DeviceClasses" Dokumente
29      // um deren Referenzen auflösen zu können
30      $unwind: {
31         path: '$classes'
32     },
33
34     {
35      // Löst die verwiesenen "HighLevel" Referenzen aus
36      // "classes.required_commands.SCPI" auf und kopiert
37      // diese Dokumente zurück in das Array
38      $lookup: {
39             from: 'HighLevel',
40             localField: 'classes.required_commands.SCPI',
41             foreignField: '_id',
42             as: 'classes.required_commands.SCPI'
43         }
44     },
45
```

```
46      {
47       // Gruppiert die Dokumente anhand der "_id"
48       // wieder in ein Dokument zusammen. Die "classes"
49       // werden dabei zu einem Array zusammengesetzt
50       $group: {
51          _id: '$_id',
52          model: {$first: '$model'},
53          manufacturer: {$first: '$manufacturer'},
54          commands: {$first: '$commands'},
55          classes: {$push: '$classes'}
56      }
57  ]
```

Quellenverzeichnis

[1] *Zentralinstitut für Engineering, Elekronik und Analytik*. abgerufen am: 5.8.2022. URL: https://www.fz-juelich.de/en/zea.

[2] *Systeme der Elektronik*. abgerufen am: 5.8.2022. URL: https://www.fz-juelich.de/en/zea/zea-2/about-us.

[3] *Interchangeable Virtual Instruments Foundation*. abgerufen am 5.7.2022. URL: https://www.ivifoundation.org.

[4] *IVI Fountaion about SCPI*. abgerufen am 20.7.2022. URL: https://www.ivifoundation.org/about/SCPI.aspx.

[5] *SCPI Consortium integration into the IVI Foundation*. abgerufen am 20.7.2022. URL: https://www.ivifoundation.org/scpi/.

[6] „IEEE Standard Digital Interface for Programmable Instrumentation“. In: *ANSI/IEEE Std 488.1-1987* (1988), S. 1–96. DOI: 10.1109/IEEESTD.1988.81527.

[7] „IEEE Standard Codes, Formats, Protocols, and Common Commands for Use With IEEE Std 488.1-1987, IEEE Standard Digital Interface for Programmable Instrumentation“. In: *IEEE Std 488.2-1992* (1992), S. 1–254. DOI: 10.1109/IEEESTD.1992.114468.

[8] SCPI Consortium. *Standard Commands for Programmable Instruments (SCPI)*. Version 1999.0. 1999. URL: https://www.ivifoundation.org/docs/SCPI-99.PDF.

[9] *What is a Graph Database? - Developer Guides*. Abgerufen am: 23.8.2022. URL: https://neo4j.com/developer/graph-database/.

[10] *Graphdatenbank – Wikipedia*. Abgerufen am: 23.8.2022. URL: https://de.wikipedia.org/wiki/Graphdatenbank.

[11] *Dokumentenorientierte Datenbanken [Wirtschaftsinformatik Wiki - Kewee]*. Abgerufen am: 23.8.2022. URL: http://wi-wiki.de/doku.php?id=bigdata:dokumentdb.

[12] *Dokumentenorientierte Datenbank – Wikipedia*. Abgerufen am: 23.8.2022. URL: https://de.wikipedia.org/wiki/Dokumentenorientierte_Datenbank.

[13] *Schema Validation - MongoDB Manual*. Abgerufen am 11.8.2022. URL: https://www.mongodb.com/docs/v6.0/core/schema-validation/.

[14] *What are ACID Properties in Database Management Systems?* abgerufen am 11.8.2022. URL: https://www.mongodb.com/basics/acid-transactions.

[15] *Role-Based Access Control*. Abgerufen am 12.8.2022. URL: https://www.mongodb.com/docs/manual/core/authorization/.

[16] *MongoDB Drivers*. Abgerufen am 11.8.2022. URL: https://www.mongodb.com/docs/drivers/.

[17] *BSON (Binary JSON) Serialization*. Abgerufen am: 12.8.2022. URL: https://bsonspec.org/.

[18] *$jsonSchema - MongoDB Manual*. Abgerufen am 22.8.2022. URL: https://www.mongodb.com/docs/manual/reference/operator/query/jsonSchema/.

[19] Francis Galiegue, Kris Zyp und Gary Court. *JSON Schema: core definitions and terminology*. Internet-Draft draft-zyp-json-schema-04. Work in Progress. Internet Engineering Task Force, Jan. 2013. 14 S. URL: https://datatracker.ietf.org/doc/draft-zyp-json-schema/04/.

Jül-4438 • Januar 2023
ISSN 0944-2952

Mitglied der Helmholtz-Gemeinschaft